廖雪峰 JavaScript Python Git 教程

wizardforcel

Published with GitBook



目錄

廖雪峰 JavaScript Python Git 教程

作者：廖雪峰

来源：廖雪峰的官方网站

赞助作者：http://www.liaoxuefeng.com/webpage/donate

JavaScript教程

这是小白的零基础JavaScript全栈教程。

JavaScript是世界上最流行的脚本语言，因为你在电脑、手机、平板上浏览的所有的网页，以及无数基于HTML5的手机App，交互逻辑都是由JavaScript驱动的。

简单地说，JavaScript是一种运行在浏览器中的解释型的编程语言。

那么问题来了，为什么我们要学JavaScript？尤其是当你已经掌握了某些其他编程语言如Java、C++的情况下。

简单粗暴的回答就是：因为你没有选择。在Web世界里，只有JavaScript能跨平台、跨浏览器驱动网页，与用户交互。

Flash背后的ActionScript曾经流行过一阵子，不过随着移动应用的兴起，没有人用Flash开发手机App，所以它目前已经边缘化了。相反，随着HTML5在PC和移动端越来越流行，JavaScript变得更加重要了。并且，新兴的Node.js把JavaScript引入到了服务器端，JavaScript已经变成了全能型选手。

JavaScript一度被认为是一种玩具编程语言，它有很多缺陷，所以不被大多数后端开发人员所重视。很多人认为，写JavaScript代码很简单，并且JavaScript只是为了在网页上添加一点交互和动画效果。

但这是完全错误的理解。JavaScript确实很容易上手，但其精髓却不为大多数开发人员所熟知。编写高质量的JavaScript代码更是难上加难。

一个合格的开发人员应该精通JavaScript和其他编程语言。如果你已经掌握了其他编程语言，或者你还什么都不会，请立刻开始学习JavaScript，不要被Web时代所淘汰。

等等，你会问道，现在有这么多在线JavaScript教程和各种从入门到精通的JavaScript书籍，为什么我要选择这个教程？

原因是，这个教程：

是JavaScript全栈教程！

可以在线免费学习！

可以在线编写JavaScript代码并直接运行！

不要再犹豫了，立刻从现在开始，零基础迈向全栈开发工程师！

JavaScript简介

JavaScript历史

要了解JavaScript，我们首先要回顾一下JavaScript的诞生。

在上个世纪的1995年，当时的网景公司正凭借其Navigator浏览器成为Web时代开启时最著名的第一代互联网公司。

由于网景公司希望能在静态HTML页面上添加一些动态效果，于是叫Brendan Eich这哥们在两周之内设计出了JavaScript语言。你没看错，这哥们只用了10天时间。

为什么起名叫JavaScript？原因是当时Java语言非常红火，所以网景公司希望借Java的名气来推广，但事实上JavaScript除了语法上有点像Java，其他部分基本上没啥关系。

ECMAScript

因为网景开发了JavaScript，一年后微软又模仿JavaScript开发了JScript，为了让JavaScript成为全球标准，几个公司联合ECMA（European Computer Manufacturers Association）组织定制了JavaScript语言的标准，被称为ECMAScript标准。

所以简单说来就是，ECMAScript是一种语言标准，而JavaScript是网景公司对ECMAScript标准的一种实现。

那为什么不直接把JavaScript定为标准呢？因为JavaScript是网景的注册商标。

不过大多数时候，我们还是用JavaScript这个词。如果你遇到ECMAScript这个词，简单把它替换为JavaScript就行了。

JavaScript版本

JavaScript语言是在10天时间内设计出来的，虽然语言的设计者水平非常NB，但谁也架不住“时间紧，任务重”，所以，JavaScript有很多设计缺陷，我们后面会慢慢讲到。

此外，由于JavaScript的标准——ECMAScript在不断发展，最新版ECMAScript 6标准（简称ES6）已经在2015年6月正式发布了，所以，讲到JavaScript的版本，实际上就是说它实现了ECMAScript标准的哪个版本。

由于浏览器在发布时就确定了JavaScript的版本，加上很多用户还在使用IE6这种古老的浏览器，这就导致你在写JavaScript的时候，要照顾一下老用户，不能一上来就用最新的ES6标准写，否则，老用户的浏览器是无法运行新版本的JavaScript代码的。

不过，JavaScript的核心语法并没有多大变化。我们的教程会先讲JavaScript最核心的用法，然后，针对ES6讲解新增特性。

快速入门

JavaScript代码可以直接嵌在网页的任何地方，不过通常我们都把JavaScript代码放到 `<head>` 中：

```
<html>
<head>
  <script>
    alert('Hello, world');
  </script>
</head>
<body>
  ...
</body>
</html>
```

由 `<script>...</script>` 包含的代码就是JavaScript代码，它将直接被浏览器执行。

第二种方法是把JavaScript代码放到一个单独的 `.js` 文件，然后在HTML中通过 `<script src="..."></script>` 引入这个文件：

```
<html>
<head>
  <script src="/static/js/abc.js"></script>
</head>
<body>
  ...
</body>
</html>
```

这样， `/static/js/abc.js` 就会被浏览器执行。

把JavaScript代码放入一个单独的 `.js` 文件中更利于维护代码，并且多个页面可以各自引用同一份 `.js` 文件。

可以在同一个页面中引入多个 `.js` 文件，还可以在页面中多次编写 `<script> js代码... </script>` ，浏览器按照顺序依次执行。

有些时候你会看到 `<script>` 标签还设置了一个 `type` 属性：

```
<script type="text/javascript">
  ...
</script>
```

但这是没有必要的，因为默认的 `type` 就是JavaScript，所以不必显式地把 `type` 指定为JavaScript。

如何编写JavaScript

可以用任何文本编辑器来编写JavaScript代码。这里我们推荐以下几种文本编辑器：

Sublime Text

免费，但不注册会不定时弹出提示框。

Notepad++

免费

注意：不可以用Word或写字板来编写JavaScript或HTML，因为带格式的文本保存后不是纯文本文件，无法被浏览器正常读取。

如何运行JavaScript

要让浏览器运行JavaScript，必须先有一个HTML页面，在HTML页面中引入JavaScript，然后，让浏览器加载该HTML页面，就可以执行JavaScript代码。

你也许会想，直接在我的硬盘上创建好HTML和JavaScript文件，然后用浏览器打开，不就可以看到效果了吗？

这种方式运行部分JavaScript代码没有问题，但由于浏览器的安全限制，以 `file://` 开头的地址无法执行如联网等JavaScript代码，最终，你还是需要架设一个Web服务器，然后以 `http://` 开头的地址来正常执行所有JavaScript代码。

不过，开始学习阶段，你无须关心如何搭建开发环境的问题，我们提供在页面输入JavaScript代码并直接运行的功能，让你专注于JavaScript的学习。

试试直接点击“Run”按钮执行下面的JavaScript代码：

```
// 以//开头直到行末的是注释，将被浏览器忽略
// 第一个JavaScript代码:

alert('Hello, world');
```

浏览器将弹出一个对话框，显示“Hello, world”。你也可以修改两个单引号中间的内容，再试着运行。

调试

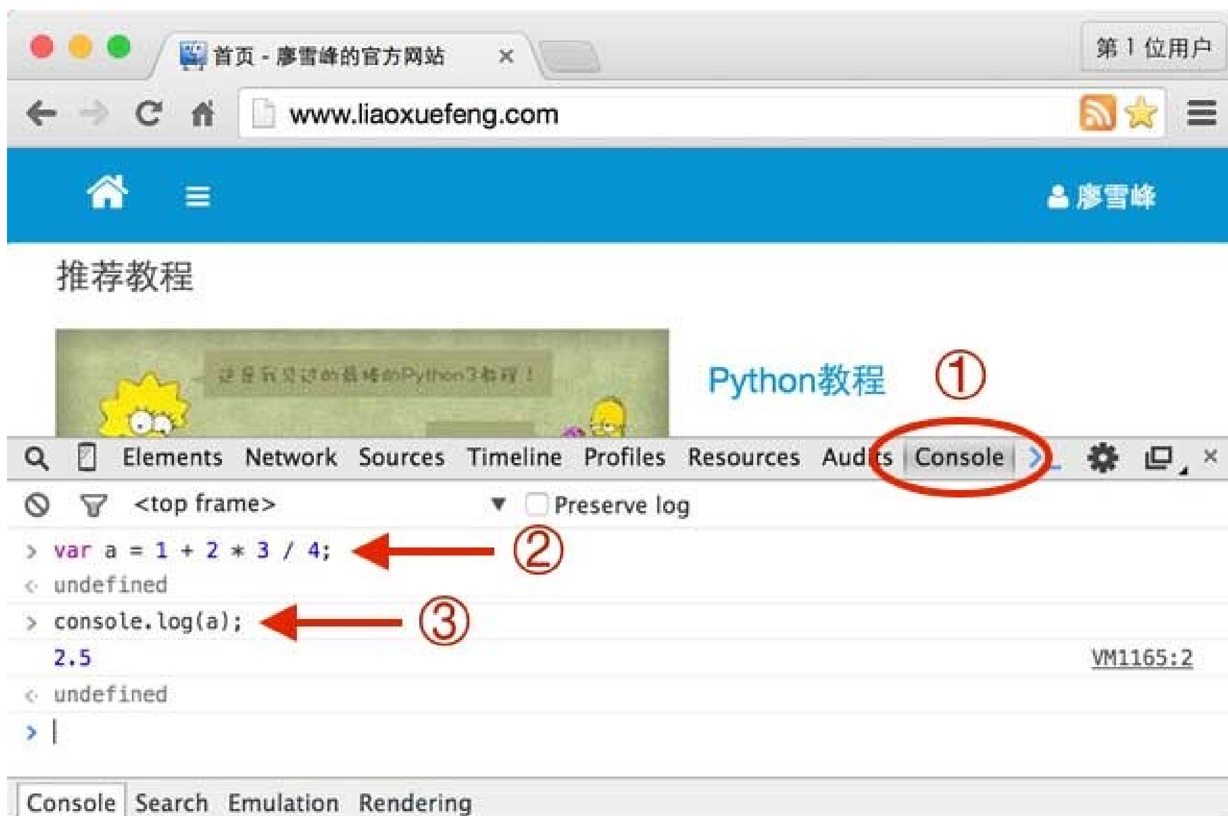
俗话说得好，“工欲善其事，必先利其器。”，写JavaScript的时候，如果期望显示 `ABC` ，结果却显示 `XYZ` ，到底代码哪里出了问题？不要抓狂，也不要泄气，作为小白，要坚信：JavaScript本身没有问题，浏览器执行也没有问题，有问题的一定是我的代码。

如何找出问题代码？这就需要调试。

怎么在浏览器中调试JavaScript代码呢？

首先，你需要安装Google Chrome浏览器，Chrome浏览器对开发者非常友好，可以让你方便地调试JavaScript代码。从这里下载Chrome浏览器。打开网页出问题的童鞋请移步国内镜像。

安装后，随便打开一个网页，然后点击菜单“查看(View)”-“开发者(Developer)”-“开发者工具(Developer Tools)”，浏览器窗口就会一分为二，下方就是开发者工具：



先点击“控制台(Console)“，在这个面板里可以直接输入JavaScript代码，按回车后执行。

要查看一个变量的内容，在Console中输入 `console.log(a);` ，回车后显示的值就是变量的内容。

关闭Console请点击右上角的“×”按钮。请熟练掌握Console的使用方法，在编写JavaScript代码时，经常需要在Console运行测试代码。

如果你对自己还有更高的要求，可以研究开发者工具的“源码(Sources)”，掌握断点、单步执行等高级调试技巧。

练习

打开新浪首页，然后查看页面源代码，找一找引入的JavaScript文件和直接编写在页面中的JavaScript代码。然后在Chrome中打开开发者工具，在控制台输入 `console.log('Hello');` ，回车查看JavaScript代码执行结果。

基本语法

语法

JavaScript的语法和Java语言类似，每个语句以 `;` 结束，语句块用 `{...}` 。但是，JavaScript并不强制要求在每个语句的结尾加 `;` ，浏览器中负责执行JavaScript代码的引擎会自动在每个语句的结尾补上 `;` 。

注意：让JavaScript引擎自动加分号在某些情况下会改变程序的语义，导致运行结果与期望不一致。在本教程中，我们不会省略 `;` ，所有语句都会添加 `;` 。

例如，下面的一行代码就是一个完整的赋值语句：

```
var x = 1;
```

下面的一行代码是一个字符串，但仍然可以视为一个完整的语句：

```
'Hello, world';
```

下面的一行代码包含两个语句，每个语句用 `;` 表示语句结束：

```
var x = 1; var y = 2; // 不建议一行写多个语句!
```

语句块是一组语句的集合，例如，下面的代码先做了一个判断，如果判断成立，将执行 `{...}` 中的所有语句：

```
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
}
```

注意花括号 `{...}` 内的语句具有缩进，通常是4个空格。缩进不是JavaScript语法要求必须的，但缩进有助于我们理解代码的层次，所以编写代码时要遵守缩进规则。很多文本编辑器具有“自动缩进”的功能，可以帮助整理代码。

`{...}` 还可以嵌套，形成层级结构：

```
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
    if (x < y) {
        z = 4;
    }
    if (x > y) {
        z = 5;
    }
}
```

JavaScript本身对嵌套的层级没有限制，但是过多的嵌套无疑会大大增加看懂代码的难度。遇到这种情况，需要把部分代码抽出来，作为函数来调用，这样可以减少代码的复杂度。

注释

以 `//` 开头直到行末的字符被视为行注释，注释是给开发人员看到，JavaScript引擎会自动忽略：

```
// 这是一行注释
alert('hello'); // 这也是注释
```

另一种块注释是用 `/*...*/` 把多行字符包裹起来，把一大“块”视为一个注释：

```
/* 从这里开始是块注释
仍然是注释
仍然是注释
注释结束 */
```

练习：

分别利用行注释和块注释把下面的语句注释掉，使它不再执行：

```
// 请注释掉下面的语句:

alert('我不想执行');
alert('我也不想执行');
```

大小写

请注意，JavaScript严格区分大小写，如果弄错了大小写，程序将报错或者运行不正常。

数据类型和变量

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在JavaScript中定义了以下几种数据类型：

Number

JavaScript不区分整数和浮点数，统一用Number表示，以下都是合法的Number类型：

```
123; // 整数123
0.456; // 浮点数0.456
1.2345e3; // 科学计数法表示1.2345x1000，等同于1234.5
-99; // 负数
NaN; // NaN表示Not a Number，当无法计算结果时用NaN表示
Infinity; // Infinity表示无限大，当数值超过了JavaScript的Number所能表示的
```

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：`0xff00`，`0xa5b4c3d2`，等等，它们和十进制表示的数值完全一样。

Number可以直接做四则运算，规则和数学一致：

```
1 + 2; // 3
(1 + 2) * 5 / 2; // 7.5
2 / 0; // Infinity
0 / 0; // NaN
10 % 3; // 1
10.5 % 3; // 1.5
```

注意 `%` 是求余运算。

字符串

字符串是以单引号'或双引号"括起来的任意文本，比如 `'abc'` ， `"xyz"` 等等。请注意， `''` 或 `""` 本身只是一种表示方式，不是字符串的一部分，因此，字符串 `'abc'` 只有 `a` ， `b` ， `c` 这3个字符。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有 `true` 、 `false` 两种值，要么是 `true` ，要么是 `false` ，可以直接用 `true` 、 `false` 表示布尔值，也可以通过布尔运算计算出来：

```
true; // 这是一个true值
false; // 这是一个false值
2 > 1; // 这是一个true值
2 >= 3; // 这是一个false值
```

`&&` 运算是与运算，只有所有都为 `true` ， `&&` 运算结果才是 `true` ：

```
true && true; // 这个&&语句计算结果为true
true && false; // 这个&&语句计算结果为false
false && true && false; // 这个&&语句计算结果为false
```

`||` 运算是或运算，只要其中有一个为 `true` ， `||` 运算结果就是 `true` ：

```
false || false; // 这个||语句计算结果为false
true || false; // 这个||语句计算结果为true
false || true || false; // 这个||语句计算结果为true
```

`!` 运算是非运算，它是一个单目运算符，把 `true` 变成 `false` ， `false` 变成 `true` ：

```
! true; // 结果为false
! false; // 结果为true
! (2 > 5); // 结果为true
```

布尔值经常用在条件判断中，比如：

```
var age = 15;
if (age >= 18) {
    alert('adult');
} else {
    alert('teenager');
}
```

比较运算符

当我们对Number做比较时，可以通过比较运算符得到一个布尔值：

```
2 > 5; // false
5 >= 2; // true
7 == 7; // true
```

实际上，JavaScript允许对任意数据类型做比较：

```
false == 0; // true
false === 0; // false
```

要特别注意相等运算符 `==` 。JavaScript在设计时，有两种比较运算符：

第一种是 `==` 比较，它会自动转换数据类型再比较，很多时候，会得到非常诡异的结果；

第二种是 `===` 比较，它不会自动转换数据类型，如果数据类型不一致，返回 `false` ，如果一致，再比较。

由于JavaScript这个设计缺陷，不要使用 `==` 比较，始终坚持使用 `===` 比较。

另一个例外是 `NaN` 这个特殊的Number与所有其他值都不相等，包括它自己：

```
NaN === NaN; // false
```

唯一能判断 `NaN` 的方法是通过 `isNaN()` 函数：

```
isNaN(NaN); // true
```

最后要注意浮点数的相等比较：

```
1 / 3 === (1 - 2 / 3); // false
```

这不是JavaScript的设计缺陷。浮点数在运算过程中会产生误差，因为计算机无法精确表示无限循环小数。要比较两个浮点数是否相等，只能计算它们之差的绝对值，看是否小于某个阈值：

```
Math.abs(1 / 3 - (1 - 2 / 3)) < 0.0000001; // true
```

null和undefined

`null` 表示一个“空”的值，它和 `0` 以及空字符串 `''` 不同， `0` 是一个数值， `''` 表示长度为0的字符串，而 `null` 表示“空”。

在其他语言中，也有类似JavaScript的 `null` 的表示，例如Java也用 `null` ，Swift用 `nil` ，Python用 `None` 表示。但是，在JavaScript中，还有一个和 `null` 类似的 `undefined` ，它表示“未定义”。

JavaScript的设计者希望用 `null` 表示一个空的值，而 `undefined` 表示值未定义。事实证明，这并没有什么卵用，区分两者的意义不大。大多数情况下，我们都应该用 `null` 。 `undefined` 仅仅在判断函数参数是否传递的情况下有用。

数组

数组是一组按顺序排列的集合，集合的每个值称为元素。JavaScript的数组可以包括任意数据类型。例如：

```
[1, 2, 3.14, 'Hello', null, true];
```

上述数组包含6个元素。数组用 `[]` 表示，元素之间用 `,` 分隔。

另一种创建数组的方法是通过 `Array()` 函数实现：

```
new Array(1, 2, 3); // 创建了数组[1, 2, 3]
```

然而，出于代码的可读性考虑，强烈建议直接使用 `[]` 。

数组的元素可以通过索引来访问。请注意，索引的起始值为 `0` ：

```
var arr = [1, 2, 3.14, 'Hello', null, true];
arr[0]; // 返回索引为0的元素，即1
arr[5]; // 返回索引为5的元素，即true
arr[6]; // 索引超出了范围，返回undefined
```

对象

JavaScript的对象是一组由键-值组成的无序集合，例如：

```
var person = {
    name: 'Bob',
    age: 20,
    tags: ['js', 'web', 'mobile'],
    city: 'Beijing',
    hasCar: true,
    zipcode: null
};
```

JavaScript对象的键都是字符串类型，值可以是任意数据类型。上述 `person` 对象一共定义了6个键值对，其中每个键又称为对象的属性，例如， `person` 的 `name` 属性为 `'Bob'` ， `zipcode` 属性为 `null` 。

要获取一个对象的属性，我们用 `对象变量.属性名` 的方式：

```
person.name; // 'Bob'
person.zipcode; // null
```

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在JavaScript中就是用一个变量名表示，变量名是大小写英文、数字、 `$` 和 `_` 的组合，且不能用数字开头。变量名也不能是JavaScript的关键字，如 `if` 、 `while` 等。申明一个变量用 `var` 语句，比如：

```
var a; // 申明了变量a，此时a的值为undefined
var $b = 1; // 申明了变量$b，同时给$b赋值，此时$b的值为1
var s_007 = '007'; // s_007是一个字符串
var Answer = true; // Answer是一个布尔值true
var t = null; // t的值是null
```

变量名也可以用中文，但是，请不要给自己找麻烦。

在JavaScript中，使用等号 `=` 对变量进行赋值。可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，但是要注意只能用 `var` 申明一次，例如：

```
var a = 123; // a的值是整数123
a = 'ABC'; // a变为字符串
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下：

```
int a = 123; // a是整数类型变量，类型用int申明
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
var x = 10;
x = x + 2;
```

如果从数学上理解 `x = x + 2` 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 `x + 2` ，得到结果 `12` ，再赋给变量 `x` 。由于 `x` 之前的值是 `10` ，重新赋值后， `x` 的值变成 `12` 。

strict模式

JavaScript在设计之初，为了方便初学者学习，并不强制要求用 `var` 申明变量。这个设计错误带来了严重的后果：如果一个变量没有通过 `var` 申明就被使用，那么该变量就自动被申明为全局变量：

```
i = 10; // i现在是全局变量
```

在同一个页面的不同的JavaScript文件中，如果都不用 `var` 申明，恰好都使用了变量 `i` ，将造成变量 `i` 互相影响，产生难以调试的错误结果。

使用 `var` 申明的变量则不是全局变量，它的范围被限制在该变量被申明的函数体内（函数的概念将稍后讲解），同名变量在不同的函数体内互不冲突。

为了修补JavaScript这一严重设计缺陷，ECMA在后续规范中推出了strict模式，在strict模式下运行的JavaScript代码，强制通过 `var` 申明变量，未使用 `var` 申明变量就使用的，将导致运行错误。

启用strict模式的方法是在JavaScript代码的第一行写上：

```
'use strict';
```

这是一个字符串，不支持strict模式的浏览器会把它当做一个字符串语句执行，支持strict模式的浏览器将开启strict模式运行JavaScript。

来测试一下你的浏览器是否能支持strict模式：

```
'use strict';
// 如果浏览器支持strict模式，
// 下面的代码将报ReferenceError错误:

abc = 'Hello, world';
alert(abc);
```

运行代码，如果浏览器报错，请修复后再运行。如果浏览器不报错，说明你的浏览器太古老了，需要尽快升级。

不用 `var` 申明的变量会被视为全局变量，为了避免这一缺陷，所有的JavaScript代码都应该使用strict模式。我们在后面编写的JavaScript代码将全部采用strict模式。

字符串

JavaScript的字符串就是用 `''` 或 `""` 括起来的字符表示。

如果 `'` 本身也是一个字符，那就可以用 `""` 括起来，比如 `"I'm OK"` 包含的字符是 `I` ， `'` ， `m` ，空格， `O` ， `K` 这6个字符。

如果字符串内部既包含 `'` 又包含 `"` 怎么办？可以用转义字符 `\` 来标识，比如：

```
'I\'m \"OK\"!';
```

表示的字符串内容是： `I'm "OK"!`

转义字符 `\` 可以转义很多字符，比如 `\n` 表示换行， `\t` 表示制表符，字符 `\` 本身也要转义，所以 `\\` 表示的字符就是 `\` 。

ASCII字符可以以 `\x##` 形式的十六进制表示，例如：

```
'\x41'; // 完全等同于 'A'
```

还可以用 `\u####` 表示一个Unicode字符：

```
'\u4e2d\u6587'; // 完全等同于 '中文'
```

由于多行字符串用 `\n` 写起来比较费事，所以最新的ES6标准新增了一种多行字符串的表示方法，用` ... `表示：

```
`这是一个
多行
字符串`;
```

练习：测试你的浏览器是否支持ES6标准，如果不支持，请把多行字符串用 `\n` 重新表示出来：

```
// 如果浏览器不支持ES6，将报SyntaxError错误:

alert(`多行
字符串
测试`);
```

字符串常见的操作如下：

```
var s = 'Hello, world!';
s.length; // 13
```

要获取字符串某个指定位置的字符，使用类似Array的下标操作，索引号从0开始：

```
var s = 'Hello, world!';

s[0]; // 'H'
s[6]; // ' '
s[7]; // 'w'
s[12]; // '!'
s[13]; // undefined 超出范围的索引不会报错，但一律返回undefined
```

需要特别注意的是，字符串是不可变的，如果对字符串的某个索引赋值，不会有任何错误，但是，也没有任何效果：

```
var s = 'Test';
s[0] = 'X';
alert(s); // s仍然为'Test'
```

JavaScript为字符串提供了一些常用方法，注意，调用这些方法本身不会改变原有字符串的内容，而是返回一个新字符串：

toUpperCase

`toUpperCase()` 把一个字符串全部变为大写：

```
var s = 'Hello';
s.toUpperCase(); // 返回'HELLO'
```

toLowerCase

`toLowerCase()` 把一个字符串全部变为小写：

```
var s = 'Hello';
var lower = s.toLowerCase(); // 返回'hello'并赋值给变量lower
lower; // 'hello'
```

indexOf

`indexOf()` 会搜索指定字符串出现的位置：

```
var s = 'hello, world';
s.indexOf('world'); // 返回7
s.indexOf('World'); // 没有找到指定的子串，返回-1
```

substring

`substring()` 返回指定索引区间的子串：

```
var s = 'hello, world'
s.substring(0, 5); // 从索引0开始到5（不包括5），返回'hello'
s.substring(7); // 从索引7开始到结束，返回'world'
```

数组

JavaScript的 `Array` 可以包含任意数据类型，并通过索引来访问每个元素。

要取得 `Array` 的长度，直接访问 `length` 属性：

```
var arr = [1, 2, 3.14, 'Hello', null, true];
arr.length; // 6
```

请注意，直接给 `Array` 的 `length` 赋一个新的值会导致 `Array` 大小的变化：

```
var arr = [1, 2, 3];
arr.length; // 3
arr.length = 6;
arr; // arr变为[1, 2, 3, undefined, undefined, undefined]
arr.length = 2;
arr; // arr变为[1, 2]
```

`Array` 可以通过索引把对应的元素修改为新的值，因此，对 `Array` 的索引进行赋值会直接修改这个 `Array` ：

```
var arr = ['A', 'B', 'C'];
arr[1] = 99;
arr; // arr现在变为['A', 99, 'C']
```

请注意，如果通过索引赋值时，索引超过了范围，同样会引起 `Array` 大小的变化：

```
var arr = [1, 2, 3];
arr[5] = 'x';
arr; // arr变为[1, 2, 3, undefined, undefined, 'x']
```

大多数其他编程语言不允许直接改变数组的大小，越界访问索引会报错。然而，JavaScript的 `Array` 却不会有任何错误。在编写代码时，不建议直接修改 `Array` 的大小，访问索引时要确保索引不会越界。

indexOf

与String类似，`Array`也可以通过`indexOf()`来搜索一个指定的元素的位置：

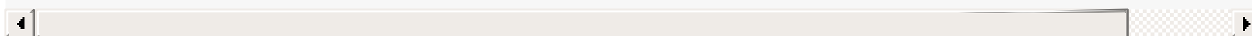
```
var arr = [10, 20, '30', 'xyz'];
arr.indexOf(10); // 元素10的索引为0
arr.indexOf(20); // 元素20的索引为1
arr.indexOf(30); // 元素30没有找到，返回-1
arr.indexOf('30'); // 元素'30'的索引为2
```

注意了，数字`30`和字符串`'30'`是不同的元素。

slice

`slice()`就是对应String的`substring()`版本，它截取`Array`的部分元素，然后返回一个新的`Array`：

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
arr.slice(0, 3); // 从索引0开始，到索引3结束，但不包括索引3: ['A', 'B',
arr.slice(3); // 从索引3开始到结束: ['D', 'E', 'F', 'G']
```



注意到`slice()`的起止参数包括开始索引，不包括结束索引。

如果不给`slice()`传递任何参数，它就会从头到尾截取所有元素。利用这一点，我们可以很容易地复制一个`Array`：

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
var aCopy = arr.slice();
aCopy; // ['A', 'B', 'C', 'D', 'E', 'F', 'G']
aCopy === arr; // false
```

push和pop

`push()`向`Array`的末尾添加若干元素，`pop()`则把`Array`的最后一个元素删除掉：

```
var arr = [1, 2];
arr.push('A', 'B'); // 返回Array新的长度: 4
arr; // [1, 2, 'A', 'B']
arr.pop(); // pop()返回'B'
arr; // [1, 2, 'A']
arr.pop(); arr.pop(); arr.pop(); // 连续pop 3次
arr; // []
arr.pop(); // 空数组继续pop不会报错，而是返回undefined
arr; // []
```

unshift和shift

如果要往 `Array` 的头部添加若干元素，使用 `unshift()` 方法， `shift()` 方法则把 `Array` 的第一个元素删掉：

```
var arr = [1, 2];
arr.unshift('A', 'B'); // 返回Array新的长度: 4
arr; // ['A', 'B', 1, 2]
arr.shift(); // 'A'
arr; // ['B', 1, 2]
arr.shift(); arr.shift(); arr.shift(); // 连续shift 3次
arr; // []
arr.shift(); // 空数组继续shift不会报错，而是返回undefined
arr; // []
```

sort

`sort()` 可以对当前 `Array` 进行排序，它会直接修改当前 `Array` 的元素位置，直接调用时，按照默认顺序排序：

```
var arr = ['B', 'C', 'A'];
arr.sort();
arr; // ['A', 'B', 'C']
```

能否按照我们自己指定的顺序排序呢？完全可以，我们将在后面的函数中讲到。

reverse

`reverse()` 把整个 `Array` 的元素给掉个个，也就是反转：

```
var arr = ['one', 'two', 'three'];
arr.reverse();
arr; // ['three', 'two', 'one']
```

splice

`splice()` 方法是修改 `Array` 的“万能方法”，它可以从指定的索引开始删除若干元素，然后再从该位置添加若干元素：

```
var arr = ['Microsoft', 'Apple', 'Yahoo', 'AOL', 'Excite', 'Oracle
// 从索引2开始删除3个元素,然后再添加两个元素:
arr.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素 ['Yahoo', '
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
// 只删除,不添加:
arr.splice(2, 2); // ['Google', 'Facebook']
arr; // ['Microsoft', 'Apple', 'Oracle']
// 只添加,不删除:
arr.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
```

concat

`concat()` 方法把当前的 `Array` 和另一个 `Array` 连接起来，并返回一个新的 `Array` ：

```
var arr = ['A', 'B', 'C'];
var added = arr.concat([1, 2, 3]);
added; // ['A', 'B', 'C', 1, 2, 3]
arr; // ['A', 'B', 'C']
```

请注意，`concat()`方法并没有修改当前`Array`，而是返回了一个新的`Array`。

实际上，`concat()`方法可以接收任意个元素和`Array`，并且自动把`Array`拆开，然后全部添加到新的`Array`里：

```
var arr = ['A', 'B', 'C'];
arr.concat(1, 2, [3, 4]); // ['A', 'B', 'C', 1, 2, 3, 4]
```

join

`join()`方法是一个非常实用的方法，它把当前`Array`的每个元素都用指定的字符串连接起来，然后返回连接后的字符串：

```
var arr = ['A', 'B', 'C', 1, 2, 3];
arr.join('-'); // 'A-B-C-1-2-3'
```

如果`Array`的元素不是字符串，将自动转换为字符串后再连接。

多维数组

如果数组的某个元素又是一个`Array`，则可以形成多维数组，例如：

```
var arr = [[1, 2, 3], [400, 500, 600], '-'];
```

上述`Array`包含3个元素，其中头两个元素本身也是`Array`。

练习：如何通过索引取到`500`这个值：

```
'use strict';
var arr = [[1, 2, 3], [400, 500, 600], '-'];

var x = ??;

alert(x); // x应该为500
```

小结

`Array` 提供了一种顺序存储一组元素的功能，并可以按索引来读写。

练习：在新生欢迎会上，你已经拿到了新同学的名单，请排序后显示：`欢迎XXX，XXX，XXX和XXX同学！`：

```
'use strict';
var arr = ['小明', '小红', '大军', '阿黄'];

alert('???');
```

对象

JavaScript的对象是一种无序的集合数据类型，它由若干键值对组成。

JavaScript的对象用于描述现实世界中的某个对象。例如，为了描述“小明”这个淘气的小朋友，我们可以用若干键值对来描述他：

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
};
```

JavaScript用一个 `{...}` 表示一个对象，键值对以 `xxx: xxx` 形式申明，用 `,` 隔开。注意，最后一个键值对不需要在末尾加 `,` ，如果加了，有的浏览器（如低版本的IE）将报错。

上述对象申明了一个 `name` 属性，值是 `'小明'` ， `birth` 属性，值是 `1990` ，以及其他一些属性。最后，把这个对象赋值给变量 `xiaoming` 后，就可以通过变量 `xiaoming` 来获取小明的属性了：

```
xiaoming.name; // '小明'
xiaoming.birth; // 1990
```

访问属性是通过 `.` 操作符完成的，但这要求属性名必须是一个有效的变量名。如果属性名包含特殊字符，就必须用 `''` 括起来：

```
var xiaohong = {
    name: '小红',
    'middle-school': 'No.1 Middle School'
};
```

`xiaohong` 的属性名 `middle-school` 不是一个有效的变量，就需要用 `''` 括起来。访问这个属性也无法使用 `.` 操作符，必须用 `['xxx']` 来访问：

```
xiaohong['middle-school']; // 'No.1 Middle School'
xiaohong['name']; // '小红'
xiaohong.name; // '小红'
```

也可以用 `xiaohong['name']` 来访问 `xiaohong` 的 `name` 属性，不过 `xiaohong.name` 的写法更简洁。我们在编写JavaScript代码的时候，属性名尽量使用标准的变量名，这样就可以直接通过 `object.prop` 的形式访问一个属性了。

实际上JavaScript对象的所有属性都是字符串，不过属性对应的值可以是任意数据类型。

如果访问一个不存在的属性会返回什么呢？JavaScript规定，访问不存在的属性不报错，而是返回 `undefined` ：

```
var xiaoming = {
    name: '小明'
};
xiaoming.age; // undefined
```

由于JavaScript的对象是动态类型，你可以自由地给一个对象添加或删除属性：

```
var xiaoming = {
    name: '小明'
};
xiaoming.age; // undefined
xiaoming.age = 18; // 新增一个age属性
xiaoming.age; // 18
delete xiaoming.age; // 删除age属性
xiaoming.age; // undefined
delete xiaoming['name']; // 删除name属性
xiaoming.name; // undefined
delete xiaoming.school; // 删除一个不存在的school属性也不会报错
```

如果我们要检测 `xiaoming` 是否拥有某一属性，可以用 `in` 操作符：

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
};
'name' in xiaoming; // true
'grade' in xiaoming; // false
```

不过要小心，如果 `in` 判断一个属性存在，这个属性不一定是 `xiaoming` 的，它可能是 `xiaoming` 继承得到的：

```
'toString' in xiaoming; // true
```

因为 `toString` 定义在 `object` 对象中，而所有对象最终都会在原型链上指向 `object` ，所以 `xiaoming` 也拥有 `toString` 属性。

要判断一个属性是否是 `xiaoming` 自身拥有的，而不是继承得到的，可以用 `hasOwnProperty()` 方法：

```
var xiaoming = {
    name: '小明'
};
xiaoming.hasOwnProperty('name'); // true
xiaoming.hasOwnProperty('toString'); // false
```

条件判断

JavaScript使用 `if () { ... } else { ... }` 来进行条件判断。例如，根据年龄显示不同内容，可以用 `if` 语句实现如下：

```
var age = 20;
if (age >= 18) { // 如果age >= 18为true，则执行if语句块
    alert('adult');
} else { // 否则执行else语句块
    alert('teenager');
}
```

其中 `else` 语句是可选的。如果语句块只包含一条语句，那么可以省略 `{}` ：

```
var age = 20;
if (age >= 18)
    alert('adult');
else
    alert('teenager');
```

省略 `{}` 的危险之处在于，如果后来想添加一些语句，却忘了写 `{}` ，就改变了 `if...else...` 的语义，例如：

```
var age = 20;
if (age >= 18)
    alert('adult');
else
    console.log('age < 18'); // 添加一行日志
    alert('teenager'); // <- 这行语句已经不在else的控制范围了
```

上述代码的 `else` 子句实际上只负责执行 `console.log('age < 18');` ，原有的 `alert('teenager');` 已经不属于 `if...else...` 的控制范围了，它每次都会执行。

相反地，有 `{}` 的语句就不会出错：

```
var age = 20;
if (age >= 18) {
    alert('adult');
} else {
    console.log('age < 18');
    alert('teenager');
}
```

这就是为什么我们建议永远都要写上 `{}` 。

多行条件判断

如果还要更细致地判断条件，可以使用多个 `if...else...` 的组合：

```
var age = 3;
if (age >= 18) {
    alert('adult');
} else if (age >= 6) {
    alert('teenager');
} else {
    alert('kid');
}
```

上述多个 `if...else...` 的组合实际上相当于两层 `if...else...` ：

```
var age = 3;
if (age >= 18) {
    alert('adult');
} else {
    if (age >= 6) {
        alert('teenager');
    } else {
        alert('kid');
    }
}
```

但是我们通常把 `else if` 连写在一起，来增加可读性。这里的 `else` 略掉了 `{}` 是没有问题的，因为它只包含一个 `if` 语句。注意最后一个单独的 `else` 不要略掉 `{}` 。

请注意， `if...else...` 语句的执行特点是二选一，在多个 `if...else...` 语句中，如果某个条件成立，则后续就不再继续判断了。

试解释为什么下面的代码显示的是 `teenager` ：

```
'use strict';
var age = 20;

if (age >= 6) {
    alert('teenager');
} else if (age >= 18) {
    alert('adult');
} else {
    alert('kid');
}
```

由于 `age` 的值为 `20` ，它实际上同时满足条件 `age >= 6` 和 `age >= 18` ，这说明条件判断的顺序非常重要。请修复后让其显示 `adult` 。

如果 `if` 的条件判断语句结果不是 `true` 或 `false` 怎么办？例如：

```
var s = '123';
if (s.length) { // 条件计算结果为3
    //
}
```

JavaScript把 `null` 、 `undefined` 、 `0` 、 `NaN` 和空字符串 `''` 视为 `false` ，其他值一概视为 `true` ，因此上述代码条件判断的结果是 `true` 。

练习

小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：

- 低于18.5：过轻
- 18.5-25：正常
- 25-28：过重
- 28-32：肥胖
- 高于32：严重肥胖

用 `if...else...` 判断并显示结果：

```
'use strict';

var height = parseFloat(prompt('请输入身高(m):'));
var weight = parseFloat(prompt('请输入体重(kg):'));

var bmi = ???;
if ...
```

循环

循环

要计算1+2+3，我们可以直接写表达式：

```
1 + 2 + 3; // 6
```

要计算1+2+3+...+10，勉强也能写出来。

但是，要计算1+2+3+...+10000，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

JavaScript的循环有两种，一种是 `for` 循环，通过初始条件、结束条件和递增条件来循环执行语句块：

```
var x = 0;
var i;
for (i=1; i<=10000; i++) {
    x = x + i;
}
x; // 50005000
```

让我们来分析一下 `for` 循环的控制条件：

- i=1 这是初始条件，将变量i置为1；
- i<=10000 这是判断条件，满足时就继续循环，不满足就退出循环；
- i++ 这是每次循环后的递增条件，由于每次循环后变量i都会加1，因此它终将在若干次循环后不满足判断条件 `i<=10000` 而退出循环。

练习

利用 `for` 循环计算 `1 * 2 * 3 * ... * 10` 的结果：

```
'use strict';

var x = ?;
var i;
for ...

if (x === 3628800) {
    alert('1 x 2 x 3 x ... x 10 = ' + x);
}
else {
    alert('计算错误');
}
```

`for` 循环最常用的地方是利用索引来遍历数组：

```
var arr = ['Apple', 'Google', 'Microsoft'];
var i, x;
for (i=0; i<arr.length; i++) {
    x = arr[i];
    alert(x);
}
```

`for` 循环的3个条件都是可以省略的，如果没有退出循环的判断条件，就必须使用 `break` 语句退出循环，否则就是死循环：

```
var x = 0;
for (;;) { // 将无限循环下去
    if (x > 100) {
        break; // 通过if判断来退出循环
    }
    x ++;
}
```

for ... in

`for` 循环的一个变体是 `for ... in` 循环，它可以把一个对象的所有属性依次循环出来：

```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    alert(key); // 'name', 'age', 'city'
}
```

要过滤掉对象继承的属性，用 `hasOwnProperty()` 来实现：

```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    if (o.hasOwnProperty(key)) {
        alert(key); // 'name', 'age', 'city'
    }
}
```

由于 `Array` 也是对象，而它的每个元素的索引被视为对象的属性，因此， `for ... in` 循环可以直接循环出 `Array` 的索引：

```
var a = ['A', 'B', 'C'];
for (var i in a) {
    alert(i); // '0', '1', '2'
    alert(a[i]); // 'A', 'B', 'C'
}
```

请注意， `for ... in` 对 `Array` 的循环得到的是 `String` 而不是 `Number` 。

while

`for` 循环在已知循环的初始和结束条件时非常有用。而上述忽略了条件的 `for` 循环容易让人看不清循环的逻辑，此时用 `while` 循环更佳。

`while` 循环只有一个判断条件，条件满足，就不断循环，条件不满足时则退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
var x = 0;
var n = 99;
while (n > 0) {
    x = x + n;
    n = n - 2;
}
x; // 2500
```

在循环内部变量 `n` 不断自减，直到变为 `-1` 时，不再满足 `while` 条件，循环退出。

do ... while

最后一种循环是 `do { ... } while()` 循环，它和 `while` 循环的唯一区别在于，不是在每次循环开始的时候判断条件，而是在每次循环完成的时候判断条件：

```
var n = 0;
do {
    n = n + 1;
} while (n < 100);
n; // 100
```

用 `do { ... } while()` 循环要小心，循环体会至少执行1次，
而 `for` 和 `while` 循环则可能一次都不执行。

练习

请利用循环遍历数组中的每个名字，并显示 `Hello, xxx!` ：

```
'use strict';
var arr = ['Bart', 'Lisa', 'Adam'];
```

请尝试`for`循环和`while`循环，并以正序、倒序两种方式遍历。

小结

循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。JavaScript的死循环会让浏览器无法正常显示或执行当前页面的逻辑，有的浏览器会直接挂掉，有的浏览器会在一段时间后提示你强行终止JavaScript的执行，因此，要特别注意死循环的问题。

在编写循环代码时，务必小心编写初始条件和判断条件，尤其是边界值。特别注意`i < 100`和`i <= 100`是不同的判断逻辑。

Map和Set

JavaScript的默认对象表示方式 `{}` 可以视为其他语言中的 `Map` 或 `Dictionary` 的数据结构，即一组键值对。

但是JavaScript的对象有个小问题，就是键必须是字符串。但实际上Number或者其他数据类型作为键也是非常合理的。

为了解决这个问题，最新的ES6规范引入了新的数据类型 `Map` 。要测试你的浏览器是否支持ES6规范，请执行以下代码，如果浏览器报ReferenceError错误，那么你需要换一个支持ES6的浏览器：

```
'use strict';
var m = new Map();
var s = new Set();
alert('你的浏览器支持Map和Set！');

// 直接运行测试
```

Map

`Map` 是一组键值对的结构，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用 `Array` 实现，需要两个 `Array` ：

```
var names = ['Michael', 'Bob', 'Tracy'];
var scores = [95, 75, 85];
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，Array越长，耗时越长。

如果用Map实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用JavaScript写一个Map如下：

```
var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);
m.get('Michael'); // 95
```

初始化 `Map` 需要一个二维数组，或者直接初始化一个空 `Map` 。 `Map` 具有以下方法：

```
var m = new Map(); // 空Map
m.set('Adam', 67); // 添加新的key-value
m.set('Bob', 59);
m.has('Adam'); // 是否存在key 'Adam': true
m.get('Adam'); // 67
m.delete('Adam'); // 删除key 'Adam'
m.get('Adam'); // undefined
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
var m = new Map();
m.set('Adam', 67);
m.set('Adam', 88);
m.get('Adam'); // 88
```

Set

`Set` 和 `Map` 类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在 `Set` 中，没有重复的key。

要创建一个 `Set` ，需要提供一个 `Array` 作为输入，或者直接创建一个空 `Set` ：

```
var s1 = new Set(); // 空Set
var s2 = new Set([1, 2, 3]); // 含1, 2, 3
```

重复元素在 `Set` 中自动被过滤：

```
var s = new Set([1, 2, 3, 3, '3']);
s; // Set {1, 2, 3, "3"}
```

注意数字 `3` 和字符串 `'3'` 是不同的元素。

通过 `add(key)` 方法可以添加元素到 `Set` 中，可以重复添加，但不会有效果：

```
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

通过 `delete(key)` 方法可以删除元素：

```
var s = new Set([1, 2, 3]);
s; // Set {1, 2, 3}
s.delete(3);
s; // Set {1, 2}
```

小结

`Map` 和 `Set` 是ES6标准新增的数据类型，请根据浏览器的支持情况决定是否要使用。

iterable

遍历`Array`可以采用下标循环，遍历`Map`和`Set`就无法使用下标。为了统一集合类型，ES6标准引入了新的`iterable`类型，`Array`、`Map`和`Set`都属于`iterable`类型。

具有`iterable`类型的集合可以通过新的`for ... of`循环来遍历。

`for ... of`循环是ES6引入的新的语法，请测试你的浏览器是否支持：

```
'use strict';
var a = [1, 2, 3];
for (var x of a) {
}
alert('你的浏览器支持for ... of');

// 请直接运行测试
```

用`for ... of`循环遍历集合，用法如下：

```
var a = ['A', 'B', 'C'];
var s = new Set(['A', 'B', 'C']);
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
for (var x of a) { // 遍历Array
    alert(x);
}
for (var x of s) { // 遍历Set
    alert(x);
}
for (var x of m) { // 遍历Map
    alert(x[0] + '=' + x[1]);
}
```

你可能会有疑问，`for ... of`循环和`for ... in`循环有何区别？

`for ... in`循环由于历史遗留问题，它遍历的实际上是对象的属性名称。一个`Array`数组实际上也是一个对象，它的每个元素的索引被视为一个属性。

当我们手动给 `Array` 对象添加了额外的属性后， `for ... in` 循环将带来意想不到的意外效果：

```
var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x in a) {
    alert(x); // '0', '1', '2', 'name'
}
```

`for ... in` 循环将把 `name` 包括在内，但 `Array` 的 `length` 属性却不包括在内。

`for ... of` 循环则完全修复了这些问题，它只循环集合本身的元素：

```
var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x of a) {
    alert(x); 'A', 'B', 'C'
}
```

这就是为什么要引入新的 `for ... of` 循环。

然而，更好的方式是直接使用 `iterable` 内置的 `forEach` 方法，它接收一个函数，每次迭代就自动回调该函数。以 `Array` 为例：

```
var a = ['A', 'B', 'C'];
a.forEach(function (element, index, array) {
    // element: 指向当前元素的值
    // index: 指向当前索引
    // array: 指向Array对象本身
    alert(element);
});
```

注意， `forEach()` 方法是ES5.1标准引入的，你需要测试浏览器是否支持。

`Set` 与 `Array` 类似，但 `Set` 没有索引，因此回调函数的前两个参数都是元素本身：

```
var s = new Set(['A', 'B', 'C']);
s.forEach(function (element, sameElement, set) {
    alert(element);
});
```

`Map` 的回调函数参数依次为 `value` 、 `key` 和 `map` 本身：

```
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
m.forEach(function (value, key, map) {
    alert(value);
});
```

如果对某些参数不感兴趣，由于JavaScript的函数调用不要求参数必须一致，因此可以忽略它们。例如，只需要获得 `Array` 的 `element` ：

```
var a = ['A', 'B', 'C'];
a.forEach(function (element) {
    alert(element);
});
```

函数

我们知道圆的面积计算公式为：

S = πr²

当我们知道半径 `r` 的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
var r1 = 12.34;
var r2 = 9.08;
var r3 = 73.1;
var s1 = 3.14 * r1 * r1;
var s2 = 3.14 * r2 * r2;
var s3 = 3.14 * r3 * r3;
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把 `3.14` 改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x`，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，JavaScript也不例外。JavaScript的函数不但是“头等公民”，而且可以像变量一样使用，具有非常强大的抽象能力。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：`1 + 2 + 3 + ... + 100`，写起来十分不方便，于是数学家发明了求和符号∑，可以把 `1 + 2 + 3 + ... + 100` 记作：

100

∑n

n=1

这种抽象记法非常强大，因为我们看到 ∑ 就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

∑(n²+1)

n=1

还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

函数定义和调用

定义函数

在JavaScript中，定义函数的方式如下：

```
function abs(x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

上述 `abs()` 函数的定义如下：

- `function` 指出这是一个函数定义；
- `abs` 是函数的名称；
- `(x)` 括号内列出函数的参数，多个参数以 `,` 分隔；
- `{ ... }` 之间的代码是函数体，可以包含若干语句，甚至可以没有任何语句。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后也会返回结果，只是结果为 `undefined` 。

由于JavaScript的函数也是一个对象，上述定义的 `abs()` 函数实际上是一个函数对象，而函数名 `abs` 可以视为指向该函数的变量。

因此，第二种定义函数的方式如下：

```
var abs = function (x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
};
```

在这种方式下， `function (x) { ... }` 是一个匿名函数，它没有函数名。但是，这个匿名函数赋值给了变量 `abs` ，所以，通过变量 `abs` 就可以调用该函数。

上述两种定义完全等价，注意第二种方式按照完整语法需要在函数体末尾加一个 `;` ，表示赋值语句结束。

调用函数

调用函数时，按顺序传入参数即可：

```
abs(10); // 返回10
abs(-9); // 返回9
```

由于JavaScript允许传入任意个参数而不影响调用，因此传入的参数比定义的参数多也没有问题，虽然函数内部并不需要这些参数：

```
abs(10, 'blablabla'); // 返回10
abs(-9, 'haha', 'hehe', null); // 返回9
```

传入的参数比定义的少也没有问题：

```
abs(); // 返回NaN
```

此时 `abs(x)` 函数的参数 `x` 将收到 `undefined` ，计算结果为 `NaN` 。

要避免收到 `undefined` ，可以对参数进行检查：

```
function abs(x) {
    if (typeof x !== 'number') {
        throw 'Not a number';
    }
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

arguments

JavaScript还有一个免费赠送的关键字 `arguments` ，它只在函数内部起作用，并且永远指向当前函数的调用者传入的所有参数。 `arguments` 类似 `Array` 但它不是一个 `Array` ：

```
function foo(x) {
    alert(x); // 10
    for (var i=0; i<arguments.length; i++) {
        alert(arguments[i]); // 10, 20, 30
    }
}
foo(10, 20, 30);
```

利用 `arguments` ，你可以获得调用者传入的所有参数。也就是说，即使函数不定义任何参数，还是可以拿到参数的值：

```
function abs() {
    if (arguments.length === 0) {
        return 0;
    }
    var x = arguments[0];
    return x >= 0 ? x : -x;
}

abs(); // 0
abs(10); // 10
abs(-9); // 9
```

实际上 `arguments` 最常用于判断传入参数的个数。你可能会看到这样的写法：

```
// foo(a[, b], c)
// 接收2~3个参数，b是可选参数，如果只传2个参数，b默认为null：
function foo(a, b, c) {
    if (arguments.length === 2) {
        // 实际拿到的参数是a和b，c为undefined
        c = b; // 把b赋给c
        b = null; // b变为默认值
    }
    // ...
}
```

要把中间的参数 `b` 变为"可选"参数，就只能通过 `arguments` 判断，然后重新调整参数并赋值。

rest参数

由于JavaScript函数允许接收任意个参数，于是我们就不得不用 `arguments` 来获取所有参数：

```
function foo(a, b) {
    var i, rest = [];
    if (arguments.length > 2) {
        for (i = 2; i<arguments.length; i++) {
            rest.push(arguments[i]);
        }
    }
    console.log('a = ' + a);
    console.log('b = ' + b);
    console.log(rest);
}
```

为了获取除了已定义参数 `a` 、 `b` 之外的参数，我们不得不用 `arguments` ，并且循环要从索引 `2` 开始以便排除前两个参数，这种写法很别扭，只是为了获得额外的 `rest` 参数，有没有更好的方法？

ES6标准引入了rest参数，上面的函数可以改写为：

```
function foo(a, b, ...rest) {
    console.log('a = ' + a);
    console.log('b = ' + b);
    console.log(rest);
}

foo(1, 2, 3, 4, 5);
// 结果:
// a = 1
// b = 2
// Array [ 3, 4, 5 ]

foo(1);
// 结果:
// a = 1
// b = undefined
// Array []
```

rest参数只能写在最后，前面用`...`标识，从运行结果可知，传入的参数先绑定`a`、`b`，多余的参数以数组形式交给变量`rest`，所以，不再需要`arguments`我们就获取了全部参数。

如果传入的参数连正常定义的参数都没填满，也不要紧，rest参数会接收一个空数组（注意不是`undefined`）。

因为rest参数是ES6新标准，所以你需要测试一下浏览器是否支持。请用rest参数编写一个`sum()`函数，接收任意个参数并返回它们的和：

```
'use strict';

function sum(...rest) {
   ???
}

// 测试:
var i, args = [];
for (i=1; i<=100; i++) {
    args.push(i);
}
if (sum() !== 0) {
    alert('测试失败: sum() = ' + sum());
} else if (sum(1) !== 1) {
    alert('测试失败: sum(1) = ' + sum(1));
} else if (sum(2, 3) !== 5) {
    alert('测试失败: sum(2, 3) = ' + sum(2, 3));
} else if (sum.apply(null, args) !== 5050) {
    alert('测试失败: sum(1, 2, 3, ..., 100) = ' + sum.apply(null, ar
} else {
    alert('测试通过!');
}
```

小心你的return语句

前面我们讲到了JavaScript引擎有一个在行末自动添加分号的机制，这可能让你栽到return语句的一个大坑：

```
function foo() {
    return { name: 'foo' };
}

foo(); // { name: 'foo' }
```

如果把return语句拆成两行：

```
function foo() {
    return
        { name: 'foo' };
}

foo(); // undefined
```

要小心了，由于JavaScript引擎在行末自动添加分号的机制，上面的代码实际上变成了：

```
function foo() {
    return; // 自动添加了分号，相当于return undefined;
        { name: 'foo' }; // 这行语句已经没法执行到了
}
```

所以正确的多行写法是：

```
function foo() {
    return { // 这里不会自动加分号，因为{表示语句尚未结束
        name: 'foo'
    };
}
```

练习

定义一个计算圆面积的函数 `area_of_circle()` ，它有两个参数：

- r: 表示圆的半径；

- pi: 表示π的值，如果不传，则默认3.14

```
'use strict';

function area_of_circle(r, pi) {

    return 0;

}
// 测试:
if (area_of_circle(2) === 12.56 && area_of_circle(2, 3.1416) === 12
    alert('测试通过');
} else {
    alert('测试失败');
}
```

Max是一个JavaScript新手，他写了一个 `max()` 函数，返回两个数中较大的那个：

```
'use strict';

function max(a, b) {

    if (a > b) {
        return
                a;
    } else {
        return
                b;
    }

}
alert(max(15, 20));
```

但是Max抱怨他的浏览器出问题了，无论传入什么数， `max()` 函数总是返回 `undefined` 。请帮他指出问题并修复。

变量作用域

在JavaScript中，用 `var` 申明的变量实际上是有作用域的。

如果一个变量在函数体内部申明，则该变量的作用域为整个函数体，在函数体外不可引用该变量：

```
'use strict';

function foo() {
    var x = 1;
    x = x + 1;
}

x = x + 2; // ReferenceError! 无法在函数体外引用变量x
```

如果两个不同的函数各自申明了同一个变量，那么该变量只在各自的函数体内起作用。换句话说，不同函数内部的同名变量互相独立，互不影响：

```
'use strict';

function foo() {
    var x = 1;
    x = x + 1;
}

function bar() {
    var x = 'A';
    x = x + 'B';
}
```

由于JavaScript的函数可以嵌套，此时，内部函数可以访问外部函数定义的变量，反过来则不行：

```
'use strict';

function foo() {
    var x = 1;
    function bar() {
        var y = x + 1; // bar可以访问foo的变量x!
    }
    var z = y + 1; // ReferenceError! foo不可以访问bar的变量y!
}
```

如果内部函数和外部函数的变量名重名怎么办？

```
'use strict';

function foo() {
    var x = 1;
    function bar() {
        var x = 'A';
        alert('x in bar() = ' + x); // 'A'
    }
    alert('x in foo() = ' + x); // 1
    bar();
}
```

这说明JavaScript的函数在查找变量时从自身函数定义开始，从“内”向“外”查找。如果内部函数定义了与外部函数重名的变量，则内部函数的变量将“屏蔽”外部函数的变量。

变量提升

JavaScript的函数定义有个特点，它会先扫描整个函数体的语句，把所有申明的变量“提升”到函数顶部：

```
'use strict';

function foo() {
    var x = 'Hello, ' + y;
    alert(x);
    var y = 'Bob';
}

foo();
```

虽然是strict模式，但语句 `var x = 'Hello, ' + y;` 并不报错，原因是变量 `y` 在稍后申明了。但是 `alert` 显示 `Hello, undefined` ，说明变量 `y` 的值为 `undefined` 。这正是因为JavaScript引擎自动提升了变量 `y` 的声明，但不会提升变量 `y` 的赋值。

对于上述 `foo()` 函数，JavaScript引擎看到的代码相当于：

```
function foo() {
    var y; // 提升变量y的申明
    var x = 'Hello, ' + y;
    alert(x);
    y = 'Bob';
}
```

由于JavaScript的这一怪异的“特性”，我们在函数内部定义变量时，请严格遵守“在函数内部首先申明所有变量”这一规则。最常见的做法是用一个 `var` 申明函数内部用到的所有变量：

```
function foo() {
    var
        x = 1, // x初始化为1
        y = x + 1, // y初始化为2
        z, i; // z和i为undefined
    // 其他语句:
    for (i=0; i<100; i++) {
        ...
    }
}
```

全局作用域

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象 `window` ，全局作用域的变量实际上被绑定到 `window` 的一个属性：

```
'use strict';

var course = 'Learn JavaScript';
alert(course); // 'Learn JavaScript'
alert(window.course); // 'Learn JavaScript'
```

因此，直接访问全局变量 `course` 和访问 `window.course` 是完全一样的。

你可能猜到了，由于函数定义有两种方式，以变量方式 `var foo = function () {}` 定义的函数实际上也是一个全局变量，因此，顶层函数的定义也被视为一个全局变量，并绑定到 `window` 对象：

```
'use strict';

function foo() {
    alert('foo');
}

foo(); // 直接调用foo()
window.foo(); // 通过window.foo()调用
```

进一步大胆地猜测，我们每次直接调用的 `alert()` 函数其实也是 `window` 的一个变量：

```
'use strict';

window.alert('调用window.alert()');
// 把alert保存到另一个变量:
var old_alert = window.alert;
// 给alert赋一个新函数:
window.alert = function () {}

alert('无法用alert()显示了!');

// 恢复alert:
window.alert = old_alert;
alert('又可以用alert()了!');
```

这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报ReferenceError错误。

名字空间

全局变量会绑定到 `window` 上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:
var MYAPP = {};

// 其他变量:
MYAPP.name = 'myapp';
MYAPP.version = 1.0;

// 其他函数:
MYAPP.foo = function () {
    return 'foo';
};
```

把自己的代码全部放入唯一的名字空间 `MYAPP` 中，会大大减少全局变量冲突的可能。

许多著名的JavaScript库都是这么干的：jQuery，YUI，underscore等等。

局部作用域

由于JavaScript的变量作用域实际上是函数内部，我们在 `for` 循环等语句块中是无法定义具有局部作用域的变量的：

```
'use strict';

function foo() {
    for (var i=0; i<100; i++) {
        //
    }
    i += 100; // 仍然可以引用变量i
}
```

为了解决块级作用域，ES6引入了新的关键字 `let` ，用 `let` 替代 `var` 可以申明一个块级作用域的变量：

```
'use strict';

function foo() {
    var sum = 0;
    for (let i=0; i<100; i++) {
        sum += i;
    }
    i += 1; // SyntaxError
}
```

常量

由于 `var` 和 `let` 申明的是变量，如果要申明一个常量，在ES6之前是不行的，我们通常用全部大写的变量来表示“这是一个常量，不要修改它的值”：

```
var PI = 3.14;
```

ES6标准引入了新的关键字 `const` 来定义常量， `const` 与 `let` 都具有块级作用域：

```
'use strict';

const PI = 3.14;
PI = 3; // 某些浏览器不报错，但是无效果！
PI; // 3.14
```

方法

在一个对象中绑定函数，称为这个对象的方法。

在JavaScript中，对象的定义是这样的：

```
var xiaoming = {
    name: '小明',
    birth: 1990
};
```

但是，如果我们给 `xiaoming` 绑定一个函数，就可以做更多的事情。比如，写个 `age()` 方法，返回 `xiaoming` 的年龄：

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    age: function () {
        var y = new Date().getFullYear();
        return y - this.birth;
    }
};

xiaoming.age; // function xiaoming.age()
xiaoming.age(); // 今年调用是25,明年调用就变成26了
```

绑定到对象上的函数称为方法，和普通函数也没啥区别，但是它在内部使用了一个 `this` 关键字，这个东东是什么？

在一个方法内部， `this` 是一个特殊变量，它始终指向当前对象，也就是 `xiaoming` 这个变量。所以， `this.birth` 可以拿到 `xiaoming` 的 `birth` 属性。

让我们拆开写：

```
function getAge() {
    var y = new Date().getFullYear();
    return y - this.birth;
}

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: getAge
};

xiaoming.age(); // 25, 正常结果
getAge(); // NaN
```

单独调用函数 `getAge()` 怎么返回了 `NaN` ？请注意，我们已经进入到了JavaScript的一个大坑里。

JavaScript的函数内部如果调用了 `this` ，那么这个 `this` 到底指向谁？

答案是，视情况而定！

如果以对象的方法形式调用，比如 `xiaoming.age()` ，该函数的 `this` 指向被调用的对象，也就是 `xiaoming` ，这是符合我们预期的。

如果单独调用函数，比如 `getAge()` ，此时，该函数的 `this` 指向全局对象，也就是 `window` 。

坑爹啊！

更坑爹的是，如果这么写：

```
var fn = xiaoming.age; // 先拿到xiaoming的age函数
fn(); // NaN
```

也是不行的！要保证 `this` 指向正确，必须用 `obj.xxx()` 的形式调用！

由于这是一个巨大的设计错误，要想纠正可没那么简单。ECMA决定，在strict模式下让函数的 `this` 指向 `undefined` ，因此，在strict模式下，你会得到一个错误：

```
'use strict';

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: function () {
        var y = new Date().getFullYear();
        return y - this.birth;
    }
};

var fn = xiaoming.age;
fn(); // Uncaught TypeError: Cannot read property 'birth' of undefi
```

这个决定只是让错误及时暴露出来，并没有解决 `this` 应该指向的正确位置。

有些时候，喜欢重构的你把方法重构了一下：

```
'use strict';

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: function () {
        function getAgeFromBirth() {
            var y = new Date().getFullYear();
            return y - this.birth;
        }
        return getAgeFromBirth();
    }
};

xiaoming.age(); // Uncaught TypeError: Cannot read property 'birth
```

结果又报错了！原因是 `this` 指针只在 `age` 方法的函数内指向 `xiaoming` ，在函数内部定义的函数， `this` 又指向 `undefined` 了！（在非strict模式下，它重新指向全局对象 `window` ！）

修复的办法也不是没有，我们用一个 `that` 变量首先捕获 `this` ：

```
'use strict';

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: function () {
        var that = this; // 在方法内部一开始就捕获this
        function getAgeFromBirth() {
            var y = new Date().getFullYear();
            return y - that.birth; // 用that而不是this
        }
        return getAgeFromBirth();
    }
};

xiaoming.age(); // 25
```

用 `var that = this;` ，你就可以放心地在方法内部定义其他函数，而不是把所有语句都堆到一个方法中。

apply

虽然在一个独立的函数调用中，根据是否是strict模式， `this` 指向 `undefined` 或 `window` ，不过，我们还是可以控制 `this` 的指向的！

要指定函数的 `this` 指向哪个对象，可以用函数本身的 `apply` 方法，它接收两个参数，第一个参数就是需要绑定的 `this` 变量，第二个参数是 `Array` ，表示函数本身的参数。

用 `apply` 修复 `getAge()` 调用：

```
function getAge() {
    var y = new Date().getFullYear();
    return y - this.birth;
}

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: getAge
};

xiaoming.age(); // 25
getAge.apply(xiaoming, []); // 25, this指向xiaoming, 参数为空
```

另一个与 `apply()` 类似的方法是 `call()` ，唯一区别是：

- `apply()` 把参数打包成 `Array` 再传入；
- `call()` 把参数按顺序传入。

比如调用 `Math.max(3, 5, 4)` ，分别用 `apply()` 和 `call()` 实现如下：

```
Math.max.apply(null, [3, 5, 4]); // 5
Math.max.call(null, 3, 5, 4); // 5
```

对普通函数调用，我们通常把 `this` 绑定为 `null` 。

装饰器

利用 `apply()` ，我们还可以动态改变函数的行为。

JavaScript的所有对象都是动态的，即使内置的函数，我们也可以重新指向新的函数。

现在假定我们想统计一下代码一共调用了多少次 `parseInt()` ，可以把所有的调用都找出来，然后手动加上 `count += 1` ，不过这样做太傻了。最佳方案是用我们自己的函数替换掉默认的 `parseInt()` ：

```
var count = 0;
var oldParseInt = parseInt; // 保存原函数

window.parseInt = function () {
    count += 1;
    return oldParseInt.apply(null, arguments); // 调用原函数
};

// 测试:
parseInt('10');
parseInt('20');
parseInt('30');
count; // 3
```

高阶函数

高阶函数英文叫Higher-order function。那么什么是高阶函数？

JavaScript的函数其实都指向某个变量。既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
function add(x, y, f) {
    return f(x) + f(y);
}
```

当我们调用 `add(-5, 6, Math.abs)` 时，参数 `x` ， `y` 和 `f` 分别接收 `-5` ， `6` 和函数 `Math.abs` ，根据函数定义，我们可以推导计算过程为：

```
x = -5;
y = 6;
f = Math.abs;
f(x) + f(y) ==> Math.abs(-5) + Math.abs(6) ==> 11;
return 11;
```

用代码验证一下：

```
add(-5, 6, Math.abs); // 11
```

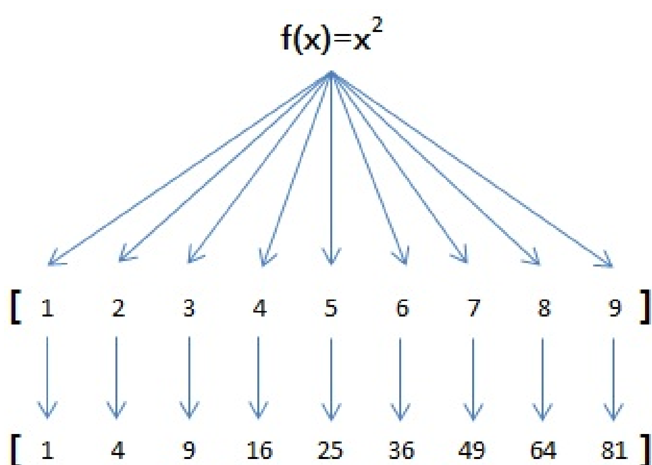
编写高阶函数，就是让函数的参数能够接收别的函数。

map/reduce

如果你读过Google的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。

map

举例说明，比如我们有一个函数f(x)=x²，要把这个函数作用在一个数组 `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 上，就可以用 `map` 实现如下：



由于 `map()` 方法定义在JavaScript的 `Array` 中，我们调用 `Array` 的 `map()` 方法，传入我们自己的函数，就得到了一个新的 `Array` 作为结果：

```
function pow(x) {
    return x * x;
}

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
arr.map(pow); // [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`map()` 传入的参数是 `pow` ，即函数对象本身。

你可能会想，不需要 `map()` ，写一个循环，也可以计算出结果：

```
var f = function (x) {
    return x * x;
};

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var result = [];
for (var i=0; i<arr.length; i++) {
    result.push(f(arr[i]));
}
```

的确可以，但是，从上面的循环代码，我们无法一眼看明白“把f(x)作用在Array的每一个元素并把结果生成一个新的Array”。

所以， `map()` 作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x²，还可以计算任意复杂的函数，比如，把 `Array` 的所有数字转为字符串：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
arr.map(String); // ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

只需要一行代码。

reduce

再看reduce的用法。Array的 `reduce()` 把一个函数作用在这个 `Array` 的 `[x1, x2, x3...]` 上，这个函数必须接收两个参数， `reduce()` 把结果继续和序列的下一个元素做累积计算，其效果就是：

```
[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)
```

比方说对一个 `Array` 求和，就可以用 `reduce` 实现：

```
var arr = [1, 3, 5, 7, 9];
arr.reduce(function (x, y) {
    return x + y;
}); // 25
```

练习：利用 `reduce()` 求积：

```
'use strict';

function product(arr) {

    return 0;

}

// 测试:
if (product([1, 2, 3, 4]) === 24 && product([0, 1, 2]) === 0 && pro
    alert('测试通过!');
}
else {
    alert('测试失败!');
}
```

要把 `[1, 3, 5, 7, 9]` 变换成整数13579，`reduce()` 也能派上用场：

```
var arr = [1, 3, 5, 7, 9];
arr.reduce(function (x, y) {
    return x * 10 + y;
}); // 13579
```

如果我们继续改进这个例子，想办法把一个字符串 `13579` 先变成 `Array` —— `[1, 3, 5, 7, 9]`，再利用 `reduce()` 就可以写出一个把字符串转换为Number的函数。

练习：不要使用JavaScript内置的 `parseInt()` 函数，利用map和reduce操作实现一个 `string2int()` 函数：

```
'use strict';

function string2int(s) {

    return 0;

}

// 测试:
if (string2int('0') === 0 && string2int('12345') === 12345 && strin
    if (string2int.toString().indexOf('parseInt') !== -1) {
        alert('请勿使用parseInt()!');
    } else if (string2int.toString().indexOf('Number') !== -1) {
        alert('请勿使用Number()!');
    } else {
        alert('测试通过!');
    }
}
else {
    alert('测试失败!');
}
```

练习

请把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：`['adam', 'LISA', 'barT']`，输出：`['Adam', 'Lisa', 'Bart']`。

```
'use strict';

function normalize(arr) {

    return [];

}

// 测试:
if (normalize(['adam', 'LISA', 'barT']).toString() === ['Adam', 'L
    alert('测试通过!');
}
else {
    alert('测试失败!');
}
```

小明希望利用 `map()` 把字符串变成整数，他写的代码很简洁：

```
'use strict';

var arr = ['1', '2', '3'];
var r;

r = arr.map(parseInt);

alert('[' + r[0] + ', ' + r[1] + ', ' + r[2] + ']');
```

结果竟然是 `[1, NaN, NaN]` ，小明百思不得其解，请帮他找到原因并修正代码。

提示：参考Array.prototype.map()的文档。

<button id="x-why-parseInt-failed" class="uk-button uk-button-success">原因分析</button>

由于 `map()` 接收的回调函数可以有3个参数： `callback(currentValue, index, array)` ，通常我们仅需要第一个参数，而忽略了传入的后面两个参数。不幸的是， `parseInt(string, radix)` 没有忽略第二个参数，导致实际执行的函数分别是：

- parseInt('0', 0); // 0, 按十进制转换
- parseInt('1', 1); // NaN, 没有一进制
- parseInt('2', 2); // NaN, 按二进制转换不允许出现2

可以改为 `r = arr.map(Number);` ，因为 `Number(value)` 函数仅接收一个参数。

filter

filter也是一个常用的操作，它用于把 `Array` 的某些元素过滤掉，然后返回剩下的元素。

和 `map()` 类似， `Array` 的 `filter()` 也接收一个函数。和 `map()` 不同的是， `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是 `true` 还是 `false` 决定保留还是丢弃该元素。

例如，在一个 `Array` 中，删掉偶数，只保留奇数，可以这么写：

```
var arr = [1, 2, 4, 5, 6, 9, 10, 15];
var r = arr.filter(function (x) {
    return x % 2 !== 0;
});
r; // [1, 5, 9, 15]
```

把一个 `Array` 中的空字符串删掉，可以这么写：

```
var arr = ['A', '', 'B', null, undefined, 'C', '  '];
var r = arr.filter(function (s) {
    return s && s.trim(); // 注意：IE9以下的版本没有trim()方法
});
r; // ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

练习

请尝试用 `filter()` 筛选出素数：

```
'use strict';

function get_primes(arr) {

    return [];

}

// 测试:
var
    x,
    r,
    arr = [];
for (x = 1; x < 100; x++) {
    arr.push(x);
}
r = get_primes(arr);
if (r.toString() === [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 4
    alert('测试通过!');
} else {
    alert('测试失败: ' + r.toString());
}
```

sort

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个对象呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 `x` 和 `y` ，如果认为 `x < y` ，则返回 `-1` ，如果认为 `x == y` ，则返回 `0` ，如果认为 `x > y` ，则返回 `1` ，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

JavaScript的 `Array` 的 `sort()` 方法就是用于排序的，但是排序结果可能让你大吃一惊：

```
// 看上去正常的结果:
['Google', 'Apple', 'Microsoft'].sort(); // ['Apple', 'Google', 'Mi

// apple排在了最后:
['Google', 'apple', 'Microsoft'].sort(); // ['Google', 'Microsoft",

// 无法理解的结果:
[10, 20, 1, 2].sort(); // [1, 10, 2, 20]
```

第二个排序把 `apple` 排在了最后，是因为字符串根据ASCII码进行排序，而小写字母 `a` 的ASCII码在大写字母之后。

第三个排序结果是什么鬼？简单的数字排序都能错？

这是因为 `Array` 的 `sort()` 方法默认把所有元素先转换为String再排序，结果 `'10'` 排在了 `'2'` 的前面，因为字符 `'1'` 比字符 `'2'` 的ASCII码小。



如果不知道 `sort()` 方法的默认排序规则，直接对数字排序，绝对栽进坑里！

幸运的是， `sort()` 方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。

要按数字大小排序，我们可以这么写：

```
var arr = [10, 20, 1, 2];
arr.sort(function (x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}); // [1, 2, 10, 20]
```

如果要倒序排序，我们可以把大的数放前面：

```
var arr = [10, 20, 1, 2];
arr.sort(function (x, y) {
    if (x < y) {
        return 1;
    }
    if (x > y) {
        return -1;
    }
    return 0;
}); // [20, 10, 2, 1]
```

默认情况下，对字符串排序，是按照ASCII的大小比较的，现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

```
var arr = ['Google', 'apple', 'Microsoft'];
arr.sort(function (s1, s2) {
    x1 = s1.toUpperCase();
    x2 = s2.toUpperCase();
    if (x1 < x2) {
        return -1;
    }
    if (x1 > x2) {
        return 1;
    }
    return 0;
}); // ['apple', 'Google', 'Microsoft']
```

忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

最后友情提示， `sort()` 方法会直接对 `Array` 进行修改，它返回的结果仍是当前 `Array` ：

```
var a1 = ['B', 'A', 'C'];
var a2 = a1.sort();
a1; // ['A', 'B', 'C']
a2; // ['A', 'B', 'C']
a1 === a2; // true, a1和a2是同一对象
```

闭包

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个对 `Array` 的求和。通常情况下，求和的函数是这样定义的：

```
function sum(arr) {
    return arr.reduce(function (x, y) {
        return x + y;
    });
}

sum([1, 2, 3, 4, 5]); // 15
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

```
function lazy_sum(arr) {
    var sum = function () {
        return arr.reduce(function (x, y) {
            return x + y;
        });
    }
    return sum;
}
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
var f = lazy_sum([1, 2, 3, 4, 5]); // function sum()
```

调用函数 `f` 时，才真正计算求和的结果：

```
f(); // 15
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum` ，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
var f1 = lazy_sum([1, 2, 3, 4, 5]);
var f2 = lazy_sum([1, 2, 3, 4, 5]);
f1 === f2; // false
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `arr` ，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push(function () {
            return i * i;
        });
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都添加到一个 `Array` 中返回了。

你可能认为调用 `f1()` ， `f2()` 和 `f3()` 结果应该是 `1` ， `4` ， `9` ，但实际结果是：

```
f1(); // 16
f2(); // 16
f3(); // 16
```

全部都是 `16` ！原因就在于返回的函数引用了变量 `i` ，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量 `i` 已经变成了 `4` ，因此最终结果为 `16` 。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push((function (n) {
            return function () {
                return n * n;
            }
        })(i));
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];

f1(); // 1
f2(); // 4
f3(); // 9
```

注意这里用了一个“创建一个匿名函数并立刻执行”的语法：

```
(function (x) {
    return x * x;
})(3); // 9
```

理论上讲，创建一个匿名函数并立刻执行可以这么写：

```
function (x) { return x * x } (3);
```

但是由于JavaScript语法解析的问题，会报SyntaxError错误，因此需要用括号把整个函数定义括起来：

```
(function (x) { return x * x }) (3);
```

通常，一个立即执行的匿名函数可以把函数体拆开，一般这么写：

```
(function (x) {
    return x * x;
})(3);
```

说了这么多，难道闭包就是为了返回一个函数然后延迟执行吗？

当然不是！闭包有非常强大的功能。举个栗子：

在面向对象的程序设计语言里，比如Java和C++，要在对象内部封装一个私有变量，可以用 `private` 修饰一个成员变量。

在没有 `class` 机制，只有函数的语言里，借助闭包，同样可以封装一个私有变量。我们用JavaScript创建一个计数器：

```
'use strict';

function create_counter(initial) {
    var x = initial || 0;
    return {
        inc: function () {
            x += 1;
            return x;
        }
    }
}
```

它用起来像这样：

```
var c1 = create_counter();
c1.inc(); // 1
c1.inc(); // 2
c1.inc(); // 3

var c2 = create_counter(10);
c2.inc(); // 11
c2.inc(); // 12
c2.inc(); // 13
```

在返回的对象中，实现了一个闭包，该闭包携带了局部变量 `x` ，并且，从外部代码根本无法访问到变量 `x` 。换句话说，闭包就是携带状态的函数，并且它的状态可以完全对外隐藏起来。

闭包还可以把多参数的函数变成单参数的函数。例如，要计算x^y可以用 `Math.pow(x, y)` 函数，不过考虑到经常计算x²或x³，我们可以利用闭包创建新的函数 `pow2` 和 `pow3` ：

```
function make_pow(n) {
    return function (x) {
        return Math.pow(x, n);
    }
}

// 创建两个新函数:
var pow2 = make_pow(2);
var pow3 = make_pow(3);

pow2(5); // 25
pow3(7); // 343
```

脑洞大开

很久很久以前，有个叫阿隆佐·邱奇的帅哥，发现只需要用函数，就可以用计算机实现运算，而不需要 `0` 、 `1` 、 `2` 、 `3` 这些数字和 `+` 、 `-` 、 `*` 、 `/` 这些符号。

JavaScript支持函数，所以可以用JavaScript用函数来写这些计算。来试试：

```
'use strict';

// 定义数字0:
var zero = function (f) {
    return function (x) {
        return x;
    }
};

// 定义数字1:
var one = function (f) {
    return function (x) {
        return f(x);
    }
};

// 定义加法:
function add(n, m) {
    return function (f) {
        return function (x) {
            return m(f)(n(f)(x));
        }
    }
}

// 计算数字2 = 1 + 1:
var two = add(one, one);

// 计算数字3 = 1 + 2:
var three = add(one, two);

// 计算数字5 = 2 + 3:
var five = add(two, three);

// 你说它是3就是3，你说它是5就是5，你怎么证明？

// 呵呵，看这里:
```

```
// 给3传一个函数,会打印3次:
(three(function () {
    console.log('print 3 times');
}))();

// 给5传一个函数,会打印5次:
(five(function () {
    console.log('print 5 times');
}))();

// 继续接着玩一会...
```

箭头函数

ES6标准新增了一种新的函数：Arraw Function（箭头函数）。

为什么叫Arrow Function？因为它的定义用的就是一个箭头：

```
x => x * x
```

上面的箭头函数相当于：

```
function (x) {
    return x * x;
}
```

在继续学习箭头函数之前，请测试你的浏览器是否支持ES6的Array Function：

```
'use strict';

var fn = x => x * x;

alert('你的浏览器支持ES6的Array Function!');
```

箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种像上面的，只包含一个表达式，连 `{ ... }` 和 `return` 都省略掉了。还有一种可以包含多条语句，这时候就不能省略 `{ ... }` 和 `return` ：

```
x => {
    if (x > 0) {
        return x * x;
    }
    else {
        return - x * x;
    }
}
```

如果参数不是一个，就需要用括号 `()` 括起来：

```
// 两个参数:
(x, y) => x * x + y * y

// 无参数:
() => 3.14

// 可变参数:
(x, y, ...rest) => {
    var i, sum = x + y;
    for (i=0; i<rest.length; i++) {
        sum += rest[i];
    }
    return sum;
}
```

如果要返回一个对象，就要注意，如果是单表达式，这么写的话会报错：

```
// SyntaxError:
x => { foo: x }
```

因为和函数体的 `{ ... }` 有语法冲突，所以要改为：

```
// ok:
x => ({ foo: x })
```

this

箭头函数看上去是匿名函数的一种简写，但实际上，箭头函数和匿名函数有个明显的区别：箭头函数内部的 `this` 是词法作用域，由上下文确定。

回顾前面的例子，由于JavaScript函数对 `this` 绑定的错误处理，下面的例子无法得到预期结果：

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = function () {
            return new Date().getFullYear() - this.birth; // this指
        };
        return fn();
    }
};
```

现在，箭头函数完全修复了 `this` 的指向， `this` 总是指向词法作用域，也就是外层调用者 `obj` ：

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = () => new Date().getFullYear() - this.birth; // th
        return fn();
    }
};
obj.getAge(); // 25
```

如果使用箭头函数，以前的那种hack写法：

```
var that = this;
```

就不再需要了。

由于 `this` 在箭头函数中已经按照词法作用域绑定了，所以，用 `call()` 或者 `apply()` 调用箭头函数时，无法对 `this` 进行绑定，即传入的第一个参数被忽略：

```
var obj = {
    birth: 1990,
    getAge: function (year) {
        var b = this.birth; // 1990
        var fn = (y) => y - this.birth; // this.birth仍是1990
        return fn.call({birth:2000}, year);
    }
};
obj.getAge(2015); // 25
```

generator

generator（生成器）是ES6标准引入的新的数据类型。一个generator看上去像一个函数，但可以返回多次。

ES6定义generator标准的哥们借鉴了Python的generator的概念和语法，如果你对Python的generator很熟悉，那么ES6的generator就是小菜一碟了。如果你对Python还不熟，赶快恶补Python教程！。

我们先复习函数的概念。一个函数是一段完整的代码，调用一个函数就是传入参数，然后返回结果：

```
function foo(x) {
    return x + x;
}

var r = foo(1); // 调用foo函数
```

函数在执行过程中，如果没有遇到 `return` 语句（函数末尾如果没有 `return`，就是隐含的 `return undefined;`），控制权无法交回被调用的代码。

generator跟函数很像，定义如下：

```
function* foo(x) {
    yield x + 1;
    yield x + 2;
    return x + 3;
}
```

generator和函数不同的是，generator由 `function*` 定义（注意多出的 `*` 号），并且，除了 `return` 语句，还可以用 `yield` 返回多次。

大多数同学立刻就晕了，generator就是能够返回多次的“函数”？返回多次有啥用？

还是举个栗子吧。

我们以一个著名的斐波那契数列为例，它由 `0`，`1` 开头：

```
0 1 1 2 3 5 8 13 21 34 ...
```

要编写一个产生斐波那契数列的函数，可以这么写：

```
function fib(max) {
    var
        t,
        a = 0,
        b = 1,
        arr = [0, 1];
    while (arr.length < max) {
        t = a + b;
        a = b;
        b = t;
        arr.push(t);
    }
    return arr;
}

// 测试:
fib(5); // [0, 1, 1, 2, 3]
fib(10); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

函数只能返回一次，所以必须返回一个 `Array` 。但是，如果换成generator，就可以一次返回一个数，不断返回多次。用generator改写如下：

```
function* fib(max) {
    var
        t,
        a = 0,
        b = 1,
        n = 1;
    while (n < max) {
        yield a;
        t = a + b;
        a = b;
        b = t;
        n ++;
    }
    return a;
}
```

直接调用试试：

```
fib(5); // fib {[[GeneratorStatus]]: "suspended", [[GeneratorReceiv
```

直接调用一个generator和调用函数不一样， `fib(5)` 仅仅是创建了一个generator对象，还没有去执行它。

调用generator对象有两个方法，一是不断地调用generator对象的 `next()` 方法：

```
var f = fib(5);
f.next(); // {value: 0, done: false}
f.next(); // {value: 1, done: false}
f.next(); // {value: 1, done: false}
f.next(); // {value: 2, done: false}
f.next(); // {value: 3, done: true}
```

`next()` 方法会执行generator的代码，然后，每次遇到 `yield x;` 就返回一个对象 `{value: x, done: true/false}` ，然后"暂停"。返回的 `value` 就是 `yield` 的返回值， `done` 表示这个generator是否已经执行结束了。如果 `done` 为 `true` ，则 `value` 就是 `return` 的返回值。

当执行到 `done` 为 `true` 时，这个generator对象就已经全部执行完毕，不要再继续调用 `next()` 了。

第二个方法是直接用 `for ... of` 循环迭代generator对象，这种方式不需要我们自己判断 `done` ：

```
for (var x of fib(5)) {
    console.log(x); // 依次输出0, 1, 1, 2, 3
}
```

generator和普通函数相比，有什么用？

因为generator可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数，利用这一点，写一个generator就可以实现需要用面向对象才能实现的功能。例如，用一个对象来保存状态，得这么写：

```
var fib = {
    a: 0,
    b: 1,
    n: 0,
    max: 5,
    next: function () {
        var
            r = this.a,
            t = this.a + this.b;
        this.a = this.b;
        this.b = t;
        if (this.n < this.max) {
            this.n ++;
            return r;
        } else {
            return undefined;
        }
    }
};
```

用对象的属性来保存状态，相当繁琐。

generator还有另一个巨大的好处，就是把异步回调代码变成“同步”代码。这个好处要等到后面学了AJAX以后才能体会到。

没有generator之前的黑暗时代，用AJAX时需要这么写代码：

```
ajax('http://url-1', data1, function (err, result) {
    if (err) {
        return handle(err);
    }
    ajax('http://url-2', data2, function (err, result) {
        if (err) {
            return handle(err);
        }
        ajax('http://url-3', data3, function (err, result) {
            if (err) {
                return handle(err);
            }
            return success(result);
        });
    });
});
```

回调越多，代码越难看。

有了generator的美好时代，用AJAX时可以这么写：

```
try {
    r1 = yield ajax('http://url-1', data1);
    r2 = yield ajax('http://url-2', data2);
    r3 = yield ajax('http://url-3', data3);
    success(r3);
}
catch (err) {
    handle(err);
}
```

看上去是同步的代码，实际执行是异步的。

练习

要生成一个自增的ID，可以编写一个 `next_id()` 函数：

```
var current_id = 0;

function next_id() {
    current_id ++;
    return current_id;
}
```

由于函数无法保存状态，故需要一个全局变量 `current_id` 来保存数字。

不用闭包，试用generator改写：

```
'use strict';
function* next_id() {



}

// 测试:
var
    x,
    pass = true,
    g = next_id();
for (x = 1; x < 100; x ++) {
    if (g.next().value !== x) {
        pass = false;
        alert('测试失败!');
        break;
    }
}
if (pass) {
    alert('测试通过!');
}
```

标准对象

在JavaScript的世界里，一切都是对象。

但是某些对象还是和其他对象不太一样。为了区分对象的类型，我们用 `typeof` 操作符获取对象的类型，它总是返回一个字符串：

```
typeof 123; // 'number'
typeof NaN; // 'number'
typeof 'str'; // 'string'
typeof true; // 'boolean'
typeof undefined; // 'undefined'
typeof Math.abs; // 'function'
typeof null; // 'object'
typeof []; // 'object'
typeof {}; // 'object'
```

可见， `number` 、 `string` 、 `boolean` 、 `function` 和 `undefined` 有别于其他类型。特别注意 `null` 的类型是 `object` ， `Array` 的类型也是 `object` ，如果我们用 `typeof` 将无法区分出 `null` 、 `Array` 和通常意义上的object —— `{}` 。

包装对象

除了这些类型外，JavaScript还提供了包装对象，熟悉Java的小伙伴肯定很清楚 `int` 和 `Integer` 这种暧昧关系。

`number` 、 `boolean` 和 `string` 都有包装对象。没错，在JavaScript中，字符串也区分 `string` 类型和它的包装类型。包装对象用 `new` 创建：

```
var n = new Number(123); // 123,生成了新的包装类型
var b = new Boolean(true); // true,生成了新的包装类型
var s = new String('str'); // 'str',生成了新的包装类型
```

虽然包装对象看上去和原来的值一模一样，显示出来也是一模一样，但他们的类型已经变为 `object` 了！所以，包装对象和原始值用 `===` 比较会返回 `false` ：

```
typeof new Number(123); // 'object'
new Number(123) === 123; // false

typeof new Boolean(true); // 'object'
new Boolean(true) === true; // false

typeof new String('str'); // 'object'
new String('str') === 'str'; // false
```

所以闲的蛋疼也不要使用包装对象！尤其是针对 `string` 类型！！！

如果我们在使用 `Number` 、 `Boolean` 和 `String` 时，没有写 `new` 会发生什么情况？

此时， `Number()` 、 `Boolean` 和 `String()` 被当做普通函数，把任何类型的数据转换为 `number` 、 `boolean` 和 `string` 类型（注意不是其包装类型）：

```
var n = Number('123'); // 123，相当于parseInt()或parseFloat()
typeof n; // 'number'

var b = Boolean('true'); // true
typeof b; // 'boolean'

var b2 = Boolean('false'); // true! 'false'字符串转换结果为true！因为它
var b3 = Boolean(''); // false

var s = String(123.45); // '123.45'
typeof s; // 'string'
```

是不是感觉头大了？这就是JavaScript特有的催眠魅力！

总结一下，有这么几条规则需要遵守：

- 不要使用 `new Number()` 、 `new Boolean()` 、 `new String()` 创建包装对象；

- 用 `parseInt()` 或 `parseFloat()` 来转换任意类型到 `number` ；
- 用 `String()` 来转换任意类型到 `string` ，或者直接调用某个对象的 `toString()` 方法；
- 通常不必把任意类型转换为 `boolean` 再判断，因为可以直接写 `if (myVar) {...}` ；
- `typeof` 操作符可以判断出 `number` 、 `boolean` 、 `string` 、 `function` 和 `undefined` ；
- 判断 `Array` 要使用 `Array.isArray(arr)` ；
- 判断 `null` 请使用 `myVar === null` ；
- 判断某个全局变量是否存在用 `typeof window.myVar === 'undefined'` ；
- 函数内部判断某个变量是否存在用 `typeof myVar === 'undefined'` 。

最后有细心的同学指出，任何对象都有 `toString()` 方法吗？ `null` 和 `undefined` 就没有！确实如此，这两个特殊值要除外，虽然 `null` 还伪装成了 `object` 类型。

更细心的同学指出， `number` 对象调用 `toString()` 报SyntaxError：

```
123.toString(); // SyntaxError
```

遇到这种情况，要特殊处理一下：

```
123..toString(); // '123', 注意是两个点！
(123).toString(); // '123'
```

不要问为什么，这就是JavaScript代码的乐趣！

Date

在JavaScript中， `Date` 对象用来表示日期和时间。

要获取系统当前时间，用：

```
var now = new Date();
now; // Wed Jun 24 2015 19:49:22 GMT+0800 (CST)
now.getFullYear(); // 2015, 年份
now.getMonth(); // 5, 月份，注意月份范围是0~11，5表示六月
now.getDate(); // 24, 表示24号
now.getDay(); // 3, 表示星期三
now.getHours(); // 19, 24小时制
now.getMinutes(); // 49, 分钟
now.getSeconds(); // 22, 秒
now.getMilliseconds(); // 875, 毫秒数
now.getTime(); // 1435146562875, 以number形式表示的时间戳
```

注意，当前时间是浏览器从本机操作系统获取的时间，所以不一定准确，因为用户可以把当前时间设定为任何值。

如果要创建一个指定日期和时间的 `Date` 对象，可以用：

```
var d = new Date(2015, 5, 19, 20, 15, 30, 123);
d; // Fri Jun 19 2015 20:15:30 GMT+0800 (CST)
```

你可能观察到了一个非常非常坑爹的地方，就是JavaScript的月份范围用整数表示是0~11， `0` 表示一月， `1` 表示二月……，所以要表示6月，我们传入的是 `5` ！这绝对是JavaScript的设计者当时脑抽了一下，但是现在要修复已经不可能了。

第二种创建一个指定日期和时间的方法是解析一个符合ISO 8601格式的字符串：

```
var d = Date.parse('2015-06-24T19:49:22.875+08:00');
d; // 1435146562875
```

但它返回的不是 `Date` 对象，而是一个时间戳。不过有时间戳就可以很容易地把它转换为一个 `Date` ：

```
var d = new Date(1435146562875);
d; // Wed Jun 24 2015 19:49:22 GMT+0800 (CST)
```

时区

`Date` 对象表示的时间总是按浏览器所在时区显示的，不过我们既可以显示本地时间，也可以显示调整后的UTC时间：

```
var d = new Date(1435146562875);
d.toLocaleString(); // '2015/6/24 下午7:49:22'，本地时间（北京时区+8:0
d.toUTCString(); // 'Wed, 24 Jun 2015 11:49:22 GMT'，UTC时间，与本地时
```

那么在JavaScript中如何进行时区转换呢？实际上，只要我们传递的是一个 `number` 类型的时间戳，我们就不用关心时区转换。任何浏览器都可以把一个时间戳正确转换为本地时间。

时间戳是个什么东西？时间戳是一个自增的整数，它表示从1970年1月1日零时整的GMT时区开始的那一刻，到现在的毫秒数。假设浏览器所在电脑的时间是准确的，那么世界上无论哪个时区的电脑，它们此刻产生的时间戳数字都是一样的，所以，时间戳可以精确地表示一个时刻，并且与时区无关。

所以，我们只需要传递时间戳，或者把时间戳从数据库里读出来，再让JavaScript自动转换为当地时间就可以了。

要获取当前时间戳，可以用：

```
if (Date.now) {
    alert(Date.now()); // 老版本IE没有now()方法
} else {
    alert(new Date().getTime());
}
```

练习

小明为了和女友庆祝情人节，特意制作了网页，并提前预定了法式餐厅。小明打算用JavaScript给女友一个惊喜留言：

```
'use strict';

var today = new Date();
if (today.getMonth() === 2 && today.getDate() === 14) {
    alert('亲爱的，我预定了晚餐，晚上6点在餐厅见！');
}
```

结果女友并未出现。小明非常郁闷，请你帮忙分析他的JavaScript代码有何问题。



RegExp

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 `@` 前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 `\d` 可以匹配一个数字， `\w` 可以匹配一个字母或数字，所以：

- `'00\d'` 可以匹配 `'007'` ，但无法匹配 `'00A'` ；
- `'\d\d\d'` 可以匹配 `'010'` ；
- `'\w\w'` 可以匹配 `'js'` ；

`.` 可以匹配任意字符，所以：

- `'js.'` 可以匹配 `'jsp'` 、 `'jss'` 、 `'js!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括0个），用 `+` 表示至少一个字符，用 `?` 表示0个或1个字符，用 `{n}` 表示n个字符，用 `{n,m}` 表示n-m个字符：

来看一个复杂的例子： `\d{3}\s+\d{3,8}` 。

我们来从左到右解读一下：

1. `\d{3}` 表示匹配3个数字，例如 `'010'` ；

2. `\s` 可以匹配一个空格（也包括Tab等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '`，`'\t\t'` 等；

3. `\d{3,8}` 表示3-8个数字，例如 `'1234567'`。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `'\'` 转义，所以，上面的正则是 `\d{3}\-\d{3,8}`。

但是，仍然无法匹配 `'010 - 12345'`，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z\_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z\_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100'`，`'0_Z'`，`'js2015'` 等等；
- `[a-zA-Z\_\$][0-9a-zA-Z\_\$]*` 可以匹配由字母或下划线、$开头，后接任意个由一个数字、字母或者下划线、$组成的字符串，也就是JavaScript允许的变量名；
- `[a-zA-Z\_\$][0-9a-zA-Z\_\$]{0, 19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `[J|j]ava[S|s]cript` 可以匹配 `'JavaScript'`、`'Javascript'`、`'javaScript'` 或者 `'javascript'`。

`^` 表示行的开头，`^\d` 表示必须以数字开头。

`$` 表示行的结束，`\d$` 表示必须以数字结束。

你可能注意到了，`js` 也可以匹配 `'jsp'`，但是加上 `^js$` 就变成了整行匹配，就只能匹配 `'js'` 了。

RegExp

有了准备知识，我们就可以在JavaScript中使用正则表达式了。

JavaScript有两种方式创建一个正则表达式：

第一种方式是直接通过 `/正则表达式/` 写出来，第二种方式是通过 `new RegExp('正则表达式')` 创建一个RegExp对象。

两种写法是一样的：

```
var re1 = /ABC\-001/;
var re2 = new RegExp('ABC\\-001');

re1; // /ABC\-001/
re2; // /ABC\-001/
```

注意，如果使用第二种写法，因为字符串的转义问题，字符串的两个 `\\` 实际上是一个 `\` 。

先看看如何判断正则表达式是否匹配：

```
var re = /^\d{3}\-\d{3,8}$/;
re.test('010-12345'); // true
re.test('010-1234x'); // false
re.test('010 12345'); // false
```

RegExp对象的 `test()` 方法用于测试给定的字符串是否符合条件。

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
'a b   c'.split(' '); // ['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
'a b   c'.split(/\s+/); // ['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入 `,` 试试：

```
'a,b, c  d'.split(/[\s\,]+/); // ['a', 'b', 'c', 'd']
```

再加入 `;` 试试：

```
'a,b;; c  d'.split(/[\s\,\;]+/); // ['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
var re = /^(\d{3})-(\d{3,8})$/;
re.exec('010-12345'); // ['010-12345', '010', '12345']
re.exec('010 12345'); // null
```

如果正则表达式中定义了组，就可以在 `RegExp` 对象上用 `exec()` 方法提取出子串来。

`exec()` 方法在匹配成功后，会返回一个 `Array` ，第一个元素始终是原始字符串本身，后面的字符串表示匹配成功的子串。

`exec()` 方法在匹配失败时返回 `null` 。

提取子串非常有用。来看一个更凶残的例子：

```
var re = /^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-
re.exec('19:05:30'); // ['19:05:30', '19', '05', '30']
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
var re = /^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9]
```

对于 `'2-30'` ，`'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0` ：

```
var re = /^(\d+)(0*)$/;
re.exec('102300'); // ['102300', '102300', '']
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。

必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
var re = /^(\d+?)(0*)$/;
re.exec('102300'); // ['102300', '1023', '00']
```

全局搜索

JavaScript的正则表达式还有几个特殊的标志，最常用的是 `g` ，表示全局匹配：

```
var r1 = /test/g;
// 等价于:
var r2 = new RegExp('test', 'g');
```

全局匹配可以多次执行 `exec()` 方法来搜索一个匹配的字符串。当我们指定 `g` 标志后，每次运行 `exec()` ，正则表达式本身会更新 `lastIndex` 属性，表示上次匹配到的最后索引：

```
var s = 'JavaScript, VBScript, JScript and ECMAScript';
var re=/[a-zA-Z]+Script/g;

// 使用全局匹配:
re.exec(s); // ['JavaScript']
re.lastIndex; // 10

re.exec(s); // ['VBScript']
re.lastIndex; // 20

re.exec(s); // ['JScript']
re.lastIndex; // 29

re.exec(s); // ['ECMAScript']
re.lastIndex; // 44

re.exec(s); // null，直到结束仍没有匹配到
```

全局匹配类似搜索，因此不能使用 `/^...$/` ，那样只会最多匹配一次。

正则表达式还可以指定 `i` 标志，表示忽略大小写， `m` 标志，表示执行多行匹配。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

```
'use strict';

var re = /^$/;

// 测试:
var
    i,
    success = true,
    should_pass = ['someone@gmail.com', 'bill.gates@microsoft.com',
    should_fail = ['test#gmail.com', 'bill@microsoft', 'bill%gates@
for (i = 0; i < should_pass.length; i++) {
    if (!re.test(should_pass[i])) {
        alert('测试失败: ' + should_pass[i]);
        success = false;
        break;
    }
}
for (i = 0; i < should_fail.length; i++) {
    if (re.test(should_fail[i])) {
        alert('测试失败: ' + should_fail[i]);
        success = false;
        break;
    }
}
if (success) {
    alert('测试通过!');
}
```

版本二可以验证并提取出带名字的Email地址：

```
'use strict';

var re = /^$/;

// 测试:
var r = re.exec('<Tom Paris> tom@voyager.org');
if (r === null || r.toString() !== ['<Tom Paris> tom@voyager.org',
    alert('测试失败!');
}
else {
    alert('测试成功!');
}
```

JSON

JSON是JavaScript Object Notation的缩写，它是一种数据交换格式。

在JSON出现之前，大家一直用XML来传递数据。因为XML是一种纯文本格式，所以它适合在网络上交换数据。XML本身不算复杂，但是，加上DTD、XSD、XPath、XSLT等一大堆复杂的规范以后，任何正常的软件开发人员碰到XML都会感觉头大了，最后大家发现，即使你努力钻研几个月，也未必搞得清楚XML的规范。

终于，在2002年的一天，道格拉斯·克罗克福特（Douglas Crockford）同学为了拯救深陷水深火热同时又被某几个巨型软件企业长期愚弄的软件工程师，发明了JSON这种超轻量级的数据交换格式。

道格拉斯同学长期担任雅虎的高级架构师，自然钟情于JavaScript。他设计的JSON实际上是JavaScript的一个子集。在JSON中，一共就这么几种数据类型：

- number：和JavaScript的 `number` 完全一致；
- boolean：就是JavaScript的 `true` 或 `false` ；
- string：就是JavaScript的 `string` ；
- null：就是JavaScript的 `null` ；
- array：就是JavaScript的 `Array` 表示方式—— `[]` ；
- object：就是JavaScript的 `{ ... }` 表示方式。

以及上面的任意组合。

并且，JSON还定死了字符集必须是UTF-8，表示多语言就没有问题了。为了统一解析，JSON的字符串规定必须用双引号 `""` ，Object的键也必须用双引号 `""` 。

由于JSON非常简单，很快就风靡Web世界，并且成为ECMA标准。几乎所有编程语言都有解析JSON的库，而在JavaScript中，我们可以直接使用JSON，因为JavaScript内置了JSON的解析。

把任何JavaScript对象变成JSON，就是把这个对象序列化成一个JSON格式的字符串，这样才能够通过网络传递给其他计算机。

如果我们收到一个JSON格式的字符串，只需要把它反序列化成一个JavaScript对象，就可以在JavaScript中直接使用这个对象了。

序列化

让我们先把小明这个对象序列化成JSON格式的字符串：

```
var xiaoming = {
    name: '小明',
    age: 14,
    gender: true,
    height: 1.65,
    grade: null,
    'middle-school': '\"W3C\" Middle School',
    skills: ['JavaScript', 'Java', 'Python', 'Lisp']
};

JSON.stringify(xiaoming); // '{"name":"小明","age":14,"gender":true
```

要输出得好看一些，可以加上参数，按缩进输出：

```
JSON.stringify(xiaoming, null, '  ');
```

结果：

```
{
  "name": "小明",
  "age": 14,
  "gender": true,
  "height": 1.65,
  "grade": null,
  "middle-school": "\"W3C\" Middle School",
  "skills": [
    "JavaScript",
    "Java",
    "Python",
    "Lisp"
  ]
}
```

第二个参数用于控制如何筛选对象的键值，如果我们只想输出指定的属性，可以传入 `Array` ：

```
JSON.stringify(xiaoming, ['name', 'skills'], '  ');
```

结果：

```
{
  "name": "小明",
  "skills": [
    "JavaScript",
    "Java",
    "Python",
    "Lisp"
  ]
}
```

还可以传入一个函数，这样对象的每个键值对都会被函数先处理：

```
function convert(key, value) {
    if (typeof value === 'string') {
        return value.toUpperCase();
    }
    return value;
}

JSON.stringify(xiaoming, convert, '  ');
```

上面的代码把所有属性值都变成大写：

```
{
  "name": "小明",
  "age": 14,
  "gender": true,
  "height": 1.65,
  "grade": null,
  "middle-school": "\"W3C\" MIDDLE SCHOOL",
  "skills": [
    "JAVASCRIPT",
    "JAVA",
    "PYTHON",
    "LISP"
  ]
}
```

如果我们还想要精确控制如何序列化小明，可以给 `xiaoming` 定义一个 `toJSON()` 的方法，直接返回JSON应该序列化的数据：

```
var xiaoming = {
    name: '小明',
    age: 14,
    gender: true,
    height: 1.65,
    grade: null,
    'middle-school': '\"W3C\" Middle School',
    skills: ['JavaScript', 'Java', 'Python', 'Lisp'],
    toJSON: function () {
        return { // 只输出name和age，并且改变了key：
            'Name': this.name,
            'Age': this.age
        };
    }
};

JSON.stringify(xiaoming); // '{"Name":"小明","Age":14}'
```

反序列化

拿到一个JSON格式的字符串，我们直接用 `JSON.parse()` 把它变成一个JavaScript对象：

```
JSON.parse('[1,2,3,true]'); // [1, 2, 3, true]
JSON.parse('{"name":"小明","age":14}'); // Object {name: '小明', age
JSON.parse('true'); // true
JSON.parse('123.45'); // 123.45
```

`JSON.parse()` 还可以接收一个函数，用来转换解析出的属性：

```
JSON.parse('{"name":"小明","age":14}', function (key, value) {
    // 把number * 2:
    if (key === 'name') {
        return value + '同学';
    }
    return value;
}); // Object {name: '小明同学', age: 14}
```

在JavaScript中使用JSON，就是这么简单！

练习

用浏览器访问Yahoo的天气API，查看返回的JSON数据。

面向对象编程

JavaScript的所有数据都可以看成对象，那是不是我们已经在使用面向对象编程了呢？

当然不是。如果我们只使用`Number`、`Array`、`string`以及基本的`{...}`定义的对象，还无法发挥出面向对象编程的威力。

JavaScript的面向对象编程和大多数其他语言如Java、C#的面向对象编程都不太一样。如果你熟悉Java或C#，很好，你一定明白面向对象的两个基本概念：

1. 类：类是对象的类型模板，例如，定义`Student`类来表示学生，类本身是一种类型，`Student`表示学生类型，但不表示任何具体的某个学生；

2. 实例：实例是根据类创建的对象，例如，根据`Student`类可以创建出`xiaoming`、`xiaohong`、`xiaojun`等多个实例，每个实例表示一个具体的学生，他们全都属于`Student`类型。

所以，类和实例是大多数面向对象编程语言的基本概念。

不过，在JavaScript中，这个概念需要改一改。JavaScript不区分类和实例的概念，而是通过原型（prototype）来实现面向对象编程。

原型是指当我们想要创建`xiaoming`这个具体的学生时，我们并没有一个`Student`类型可用。那怎么办？恰好有这么一个现成的对象：

```
var robot = {
    name: 'Robot',
    height: 1.6,
    run: function () {
        console.log(this.name + ' is running...');
    }
};
```

我们看这个`robot`对象有名字，有身高，还会跑，有点像小明，干脆就根据它来“创建”小明得了！

于是我们把它改名为`Student`，然后创建出`xiaoming`：

```
var Student = {
    name: 'Robot',
    height: 1.2,
    run: function () {
        console.log(this.name + ' is running...');
    }
};

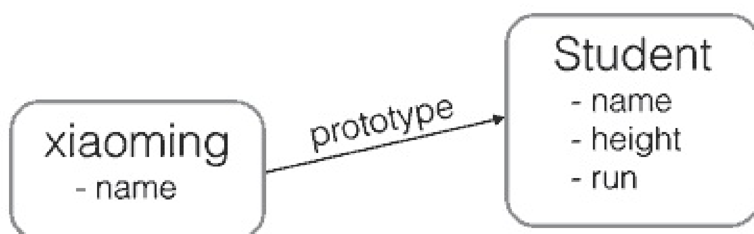
var xiaoming = {
    name: '小明'
};

xiaoming.__proto__ = Student;
```

注意最后一行代码把 `xiaoming` 的原型指向了对象 `Student` ，看上去 `xiaoming` 仿佛是从 `Student` 继承下来的：

```
xiaoming.name; // '小明'
xiaoming.run(); // 小明 is running...
```

`xiaoming` 有自己的 `name` 属性，但并没有定义 `run()` 方法。不过，由于小明是从 `Student` 继承而来，只要 `Student` 有 `run()` 方法， `xiaoming` 也可以调用：



JavaScript的原型链和Java的Class区别就在，它没有“Class”的概念，所有对象都是实例，所谓继承关系不过是把一个对象的原型指向另一个对象而已。

如果你把 `xiaoming` 的原型指向其他对象：

```
var Bird = {
    fly: function () {
        console.log(this.name + ' is flying...');
    }
};

xiaoming.__proto__ = Bird;
```

现在 `xiaoming` 已经无法 `run()` 了，他已经变成了一只鸟：

```
xiaoming.fly(); // 小明 is flying...
```

在JavaScrip代码运行时期，你可以把 `xiaoming` 从 `Student` 变成 `Bird` ，或者变成任何对象。

请注意，上述代码仅用于演示目的。在编写JavaScript代码时，不要直接用 `obj.__proto__` 去改变一个对象的原型，并且，低版本的IE也无法使用 `__proto__` 。 `Object.create()` 方法可以传入一个原型对象，并创建一个基于该原型的新对象，但是新对象什么属性都没有，因此，我们可以编写一个函数来创建 `xiaoming` ：

```
// 原型对象:
var Student = {
    name: 'Robot',
    height: 1.2,
    run: function () {
        console.log(this.name + ' is running...');
    }
};

function createStudent(name) {
    // 基于Student原型创建一个新对象:
    var s = Object.create(Student);
    // 初始化新对象:
    s.name = name;
    return s;
}

var xiaoming = createStudent('小明');
xiaoming.run(); // 小明 is running...
xiaoming.__proto__ === Student; // true
```

这是创建原型继承的一种方法，JavaScript还有其他方法来创建对象，我们在后面会一一讲到。

创建对象

JavaScript对每个创建的对象都会设置一个原型，指向它的原型对象。

当我们用 `obj.xxx` 访问一个对象的属性时，JavaScript引擎先在当前对象上查找该属性，如果没有找到，就到其原型对象上找，如果还没有找到，就一直上溯到 `Object.prototype` 对象，最后，如果还没有找到，就只能返回 `undefined` 。

例如，创建一个 `Array` 对象：

```
var arr = [1, 2, 3];
```

其原型链是：

```
arr ----> Array.prototype ----> Object.prototype ----> null
```

`Array.prototype` 定义了 `indexOf()` 、 `shift()` 等方法，因此你可以在所有的 `Array` 对象上直接调用这些方法。

当我们创建一个函数时：

```
function foo() {
    return 0;
}
```

函数也是一个对象，它的原型链是：

```
foo ----> Function.prototype ----> Object.prototype ----> null
```

由于 `Function.prototype` 定义了 `apply()` 等方法，因此，所有函数都可以调用 `apply()` 方法。

很容易想到，如果原型链很长，那么访问一个对象的属性就会因为花更多的时间查找而变得更慢，因此要注意不要把原型链搞得太长。

构造函数

除了直接用 `{ ... }` 创建一个对象外，JavaScript还可以用一种构造函数的方法来创建对象。它的用法是，先定义一个构造函数：

```
function Student(name) {
    this.name = name;
    this.hello = function () {
        alert('Hello, ' + this.name + '!');
    }
}
```

你会问，咦，这不是一个普通函数吗？

这确实是一个普通函数，但是在JavaScript中，可以用关键字 `new` 来调用这个函数，并返回一个对象：

```
var xiaoming = new Student('小明');
xiaoming.name; // '小明'
xiaoming.hello(); // Hello, 小明!
```

注意，如果不写 `new` ，这就是一个普通函数，它返回 `undefined` 。但是，如果写了 `new` ，它就变成了一个构造函数，它绑定的 `this` 指向新创建的对象，并默认返回 `this` ，也就是说，不需要在最后写 `return this;` 。

新创建的 `xiaoming` 的原型链是：

```
xiaoming ----> Student.prototype ----> Object.prototype ----> null
```

也就是说， `xiaoming` 的原型指向函数 `Student` 的原型。如果你又创建了 `xiaohong` 、 `xiaojun` ，那么这些对象的原型与 `xiaoming` 是一样的：

```
xiaoming ↘
xiaohong -→ Student.prototype ----> Object.prototype ----> null
xiaojun  ↗
```

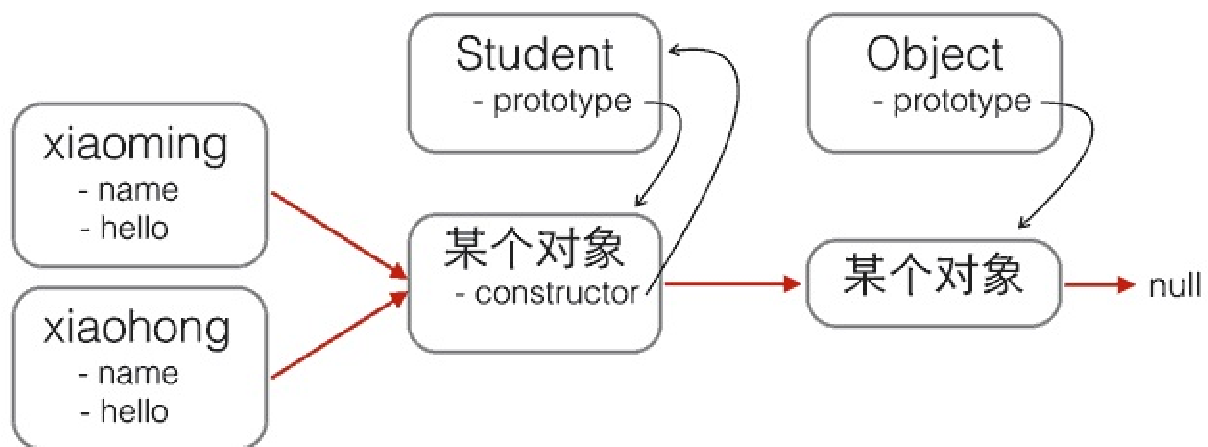
用 `new Student()` 创建的对象还从原型上获得了一个 `constructor` 属性，它指向函数 `Student` 本身：

```
xiaoming.constructor === Student.prototype.constructor; // true
Student.prototype.constructor === Student; // true

Object.getPrototypeOf(xiaoming) === Student.prototype; // true

xiaoming instanceof Student; // true
```

看晕了吧？用一张图来表示这些乱七八糟的关系就是：



红色箭头是原型链。注意，`Student.prototype` 指向的对象就是 `xiaoming`、`xiaohong` 的原型对象，这个原型对象自己还有个属性 `constructor`，指向 `Student` 函数本身。

另外，函数 `Student` 恰好有个属性 `prototype` 指向 `xiaoming`、`xiaohong` 的原型对象，但是 `xiaoming`、`xiaohong` 这些对象可没有 `prototype` 这个属性，不过可以用 `__proto__` 这个非标准用法来查看。

现在我们就认为 `xiaoming`、`xiaohong` 这些对象“继承”自 `Student`。

不过还有一个小问题，注意观察：

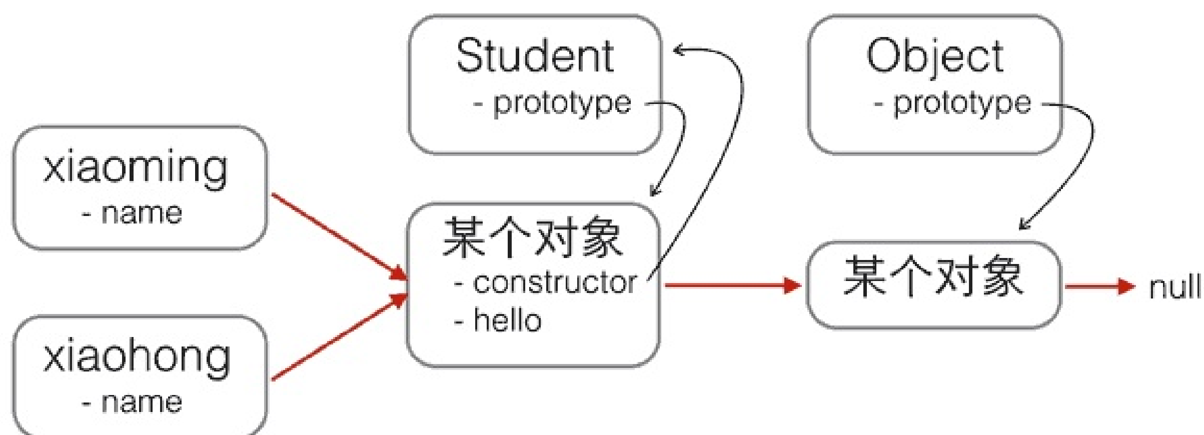
```
xiaoming.name; // '小明'
xiaohong.name; // '小红'
xiaoming.hello; // function: Student.hello()
xiaohong.hello; // function: Student.hello()
xiaoming.hello === xiaohong.hello; // false
```

`xiaoming` 和 `xiaohong` 各自的 `name` 不同，这是对的，否则我们无法区分谁是谁了。

`xiaoming` 和 `xiaohong` 各自的 `hello` 是一个函数，但它们是两个不同的函数，虽然函数名称和代码都是相同的！

如果我们通过 `new Student()` 创建了很多对象，这些对象的 `hello` 函数实际上只需要共享同一个函数就可以了，这样可以节省很多内存。

要让创建的对象共享一个 `hello` 函数，根据对象的属性查找原则，我们只要把 `hello` 函数移动到 `xiaoming` 、 `xiaohong` 这些对象共同的原型上就可以了，也就是 `Student.prototype` ：



修改代码如下：

```
function Student(name) {
    this.name = name;
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
};
```

用 `new` 创建基于原型的JavaScript的对象就是这么简单！

忘记写new怎么办

如果一个函数被定义为用于创建对象的构造函数，但是调用时忘记了写 `new` 怎么办？

在strict模式下， `this.name = name` 将报错，因为 `this` 绑定为 `undefined` ，在非strict模式下， `this.name = name` 不报错，因为 `this` 绑定为 `window` ，于是无意间创建了全局变量 `name` ，并且返回 `undefined` ，这个结果更糟糕。

所以，调用构造函数千万不要忘记写 `new` 。为了区分普通函数和构造函数，按照约定，构造函数首字母应当大写，而普通函数首字母应当小写，这样，一些语法检查工具如jslint将可以帮你检测到漏写的 `new` 。

最后，我们还可以编写一个 `createStudent()` 函数，在内部封装所有的 `new` 操作。一个常用的编程模式像这样：

```
function Student(props) {
    this.name = props.name || '匿名'; // 默认值为'匿名'
    this.grade = props.grade || 1; // 默认值为1
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
};

function createStudent(props) {
    return new Student(props || {})
}
```

这个 `createStudent()` 函数有几个巨大的优点：一是不需要 `new` 来调用，二是参数非常灵活，可以不传，也可以这么传：

```
var xiaoming = createStudent({
    name: '小明'
});

xiaoming.grade; // 1
```

如果创建的对象有很多属性，我们只需要传递需要的某些属性，剩下的属性可以用默认值。由于参数是一个Object，我们无需记忆参数的顺序。如果恰好从 `JSON` 拿到了一个对象，就可以直接创建出 `xiaoming` 。

练习

请利用构造函数定义 `Cat` ，并让所有的Cat对象有一个 `name` 属性，并共享一个方法 `say()` ，返回字符串 `'Hello, xxx!'` ：

```
'use strict';

function Cat(name) {
    //
}

// 测试:
var kitty = new Cat('Kitty');
var doraemon = new Cat('哆啦A梦');
if (kitty && kitty.name === 'Kitty' && kitty.say && typeof kitty.sa
    alert('测试通过!');
} else {
    alert('测试失败!');
}
```

原型继承

在传统的基于Class的语言如Java、C++中，继承的本质是扩展一个已有的Class，并生成新的Subclass。

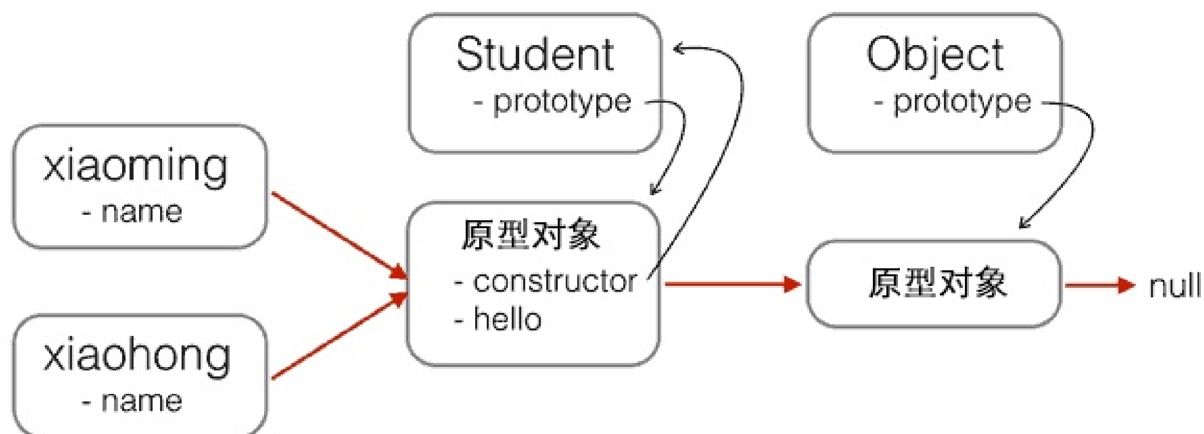
由于这类语言严格区分类和实例，继承实际上是类型的扩展。但是，JavaScript由于采用原型继承，我们无法直接扩展一个Class，因为根本不存在Class这种类型。

但是办法还是有的。我们先回顾 `Student` 构造函数：

```
function Student(props) {
    this.name = props.name || 'Unnamed';
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
}
```

以及 `Student` 的原型链：



现在，我们要基于 `Student` 扩展出 `PrimaryStudent` ，可以先定义出 `PrimaryStudent` ：

```
function PrimaryStudent(props) {
    // 调用Student构造函数，绑定this变量:
    Student.call(this, props);
    this.grade = props.grade || 1;
}
```

但是，调用了 `Student` 构造函数不等于继承了 `Student` ， `PrimaryStudent` 创建的对象的原型是：

```
new PrimaryStudent() ----> PrimaryStudent.prototype ----> Object.pr
```

必须想办法把原型链修改为：

```
new PrimaryStudent() ----> PrimaryStudent.prototype ----> Student.p
```

这样，原型链对了，继承关系就对了。新的基于 `PrimaryStudent` 创建的对象不但能调用 `PrimaryStudent.prototype` 定义的方法，也可以调用 `Student.prototype` 定义的方法。

如果你想用最简单粗暴的方法这么干：

```
PrimaryStudent.prototype = Student.prototype;
```

是不行的！如果这样的话， `PrimaryStudent` 和 `Student` 共享一个原型对象，那还要定义 `PrimaryStudent` 干啥？

我们必须借助一个中间对象来实现正确的原型链，这个中间对象的原型要指向 `Student.prototype` 。为了实现这一点，参考道爷（就是发明JSON的那个道格拉斯）的代码，中间对象可以用一个空函数 `F` 来实现：

```
// PrimaryStudent构造函数:
function PrimaryStudent(props) {
    Student.call(this, props);
    this.grade = props.grade || 1;
}
```

```
// 空函数F:
function F() {
}

// 把F的原型指向Student.prototype:
F.prototype = Student.prototype;

// 把PrimaryStudent的原型指向一个新的F对象，F对象的原型正好指向Student.pro
PrimaryStudent.prototype = new F();

// 把PrimaryStudent原型的构造函数修复为PrimaryStudent:
PrimaryStudent.prototype.constructor = PrimaryStudent;

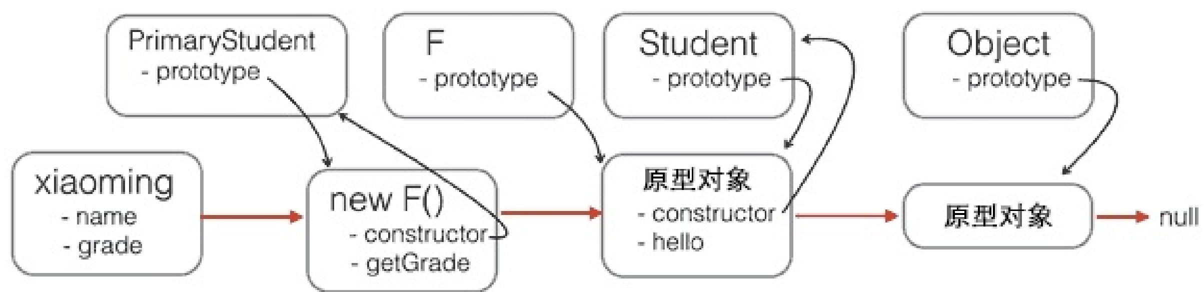
// 继续在PrimaryStudent原型（就是new F()对象）上定义方法：
PrimaryStudent.prototype.getGrade = function () {
    return this.grade;
};

// 创建xiaoming:
var xiaoming = new PrimaryStudent({
    name: '小明',
    grade: 2
});
xiaoming.name; // '小明'
xiaoming.grade; // 2

// 验证原型:
xiaoming.__proto__ === PrimaryStudent.prototype; // true
xiaoming.__proto__.__proto__ === Student.prototype; // true

// 验证继承关系:
xiaoming instanceof PrimaryStudent; // true
xiaoming instanceof Student; // true
```

用一张图来表示新的原型链：



注意，函数 `F` 仅用于桥接，我们仅创建了一个 `new F()` 实例，而且，没有改变原有的 `Student` 定义的原型链。

如果把继承这个动作用一个 `inherits()` 函数封装起来，还可以隐藏 `F` 的定义，并简化代码：

```
function inherits(Child, Parent) {
    var F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
}
```

这个 `inherits()` 函数可以复用：

```
function Student(props) {
    this.name = props.name || 'Unnamed';
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
}

function PrimaryStudent(props) {
    Student.call(this, props);
    this.grade = props.grade || 1;
}

// 实现原型继承链:
inherits(PrimaryStudent, Student);

// 绑定其他方法到PrimaryStudent原型:
PrimaryStudent.prototype.getGrade = function () {
    return this.grade;
};
```

小结

JavaScript的原型继承实现方式就是：

1. 定义新的构造函数，并在内部用 `call()` 调用希望“继承”的构造函数，并绑定 `this` ；
2. 借助中间函数 `F` 实现原型链继承，最好通过封装的 `inherits` 函数完成；
3. 继续在新的构造函数的原型上定义新方法。

浏览器

由于JavaScript的出现就是为了能在浏览器中运行，所以，浏览器自然是JavaScript开发者必须要关注的。

目前主流的浏览器分这么几种：

- IE 6~11：国内用得最多的IE浏览器，历来对W3C标准支持差。从IE10开始支持ES6标准；
- Chrome：Google出品的基于Webkit内核浏览器，内置了非常强悍的JavaScript引擎——V8。由于Chrome一经安装就时刻保持自升级，所以不用管它的版本，最新版早就支持ES6了；
- Sarafi：Apple的Mac系统自带的基于Webkit内核的浏览器，从OS X 10.7 Lion自带的6.1版本开始支持ES6，目前最新的OS X 10.10 Yosemite自带的Sarafi版本是8.x，早已支持ES6；
- Firefox：Mozilla自己研制的Gecko内核和JavaScript引擎OdinMonkey。早期的Firefox按版本发布，后来终于聪明地学习Chrome的做法进行自升级，时刻保持最新；
- 移动设备上目前iOS和Android两大阵营分别主要使用Apple的Safari和Google的Chrome，由于两者都是Webkit核心，结果HTML5首先在手机上全面普及（桌面绝对是Microsoft拖了后腿），对JavaScript的标准支持也很好，最新版本均支持ES6。

其他浏览器如Opera等由于市场份额太小就被自动忽略了。

另外还要注意识别各种国产浏览器，如某某安全浏览器，某某旋风浏览器，它们只是做了一个壳，其核心调用的是IE，也有号称同时支持IE和Webkit的“双核”浏览器。

不同的浏览器对JavaScript支持的差异主要是，有些API的接口不一样，比如AJAX，File接口。对于ES6标准，不同的浏览器对各个特性支持也不一样。

在编写JavaScript的时候，就要充分考虑到浏览器的差异，尽量让同一份JavaScript代码能运行在不同的浏览器中。

浏览器对象

JavaScript可以获取浏览器提供的很多对象，并进行操作。

window

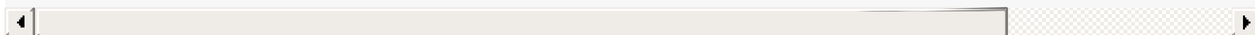
`window` 对象不但充当全局作用域，而且表示浏览器窗口。

`window` 对象有 `innerWidth` 和 `innerHeight` 属性，可以获取浏览器窗口的内部宽度和高度。内部宽高是指除去菜单栏、工具栏、边框等占位元素后，用于显示网页的净宽高。

兼容性：IE<=8不支持。

```
'use strict';

// 可以调整浏览器窗口大小试试:
alert('window inner size: ' + window.innerWidth + ' x ' + window.in
```



对应的，还有一个 `outerWidth` 和 `outerHeight` 属性，可以获取浏览器窗口的整个宽高。

navigator

`navigator` 对象表示浏览器的信息，最常用的属性包括：

- navigator.appName：浏览器名称；
- navigator.appVersion：浏览器版本；
- navigator.language：浏览器设置的语言；
- navigator.platform：操作系统类型；
- navigator.userAgent：浏览器设定的 `User-Agent` 字符串。

```
'use strict';

alert('appName = ' + navigator.appName + '\n' +
      'appVersion = ' + navigator.appVersion + '\n' +
      'language = ' + navigator.language + '\n' +
      'platform = ' + navigator.platform + '\n' +
      'userAgent = ' + navigator.userAgent);
```

请注意，`navigator` 的信息可以很容易地被用户修改，所以JavaScript读取的值不一定是正确的。很多初学者为了针对不同浏览器编写不同的代码，喜欢用 `if` 判断浏览器版本，例如：

```
var width;
if (getIEVersion(navigator.userAgent) < 9) {
    width = document.body.clientWidth;
} else {
    width = window.innerWidth;
}
```

但这样既可能判断不准确，也很难维护代码。正确的方法是充分利用JavaScript对不存在属性返回 `undefined` 的特性，直接用短路运算符 `||` 计算：

```
var width = window.innerWidth || document.body.clientWidth;
```

screen

`screen` 对象表示屏幕的信息，常用的属性有：

- screen.width：屏幕宽度，以像素为单位；
- screen.height：屏幕高度，以像素为单位；
- screen.colorDepth：返回颜色位数，如8、16、24。

```
'use strict';

alert('Screen size = ' + screen.width + ' x ' + screen.height);
```

location

`location` 对象表示当前页面的URL信息。例如，一个完整的URL：

```
http://www.example.com:8080/path/index.html?a=1&b=2#TOP
```

可以用 `location.href` 获取。要获得URL各个部分的值，可以这么写：

```
location.protocol; // 'http'
location.host; // 'www.example.com'
location.port; // '8080'
location.pathname; // '/path/index.html'
location.search; // '?a=1&b=2'
location.hash; // 'TOP'
```

要加载一个新页面，可以调用 `location.assign()` 。如果要重新加载当前页面，调用 `location.reload()` 方法非常方便。

```
'use strict';

if (confirm('重新加载当前页' + location.href + '?')) {
    location.reload();
} else {
    location.assign('/discuss'); // 设置一个新的URL地址
}
```

document

`document` 对象表示当前页面。由于HTML在浏览器中以DOM形式表示为树形结构， `document` 对象就是整个DOM树的根节点。

`document` 的 `title` 属性是从HTML文档中的 `<title>xxx</title>` 读取的，但是可以动态改变：

```
'use strict';

document.title = '努力学习JavaScript!';
```

请观察浏览器窗口标题的变化。

要查找DOM树的某个节点，需要从 `document` 对象开始查找。最常用的查找是根据ID和Tag Name。

我们先准备HTML数据：

```
<dl id="drink-menu" style="border:solid 1px #ccc;padding:6px;">
    <dt>摩卡</dt>
    <dd>热摩卡咖啡</dd>
    <dt>酸奶</dt>
    <dd>北京老酸奶</dd>
    <dt>果汁</dt>
    <dd>鲜榨苹果汁</dd>
</dl>
```

用 `document` 对象提供的 `getElementById()` 和 `getElementsByTagName()` 可以按ID获得一个DOM节点和按Tag名称获得一组DOM节点：

```
'use strict';

var menu = document.getElementById('drink-menu');
var drinks = document.getElementsByTagName('dt');
var i, s, menu, drinks;

menu = document.getElementById('drink-menu');
menu.tagName; // 'DL'

drinks = document.getElementsByTagName('dt');
s = '提供的饮料有:';
for (i=0; i<drinks.length; i++) {
    s = s + drinks[i].innerHTML + ',';
}
alert(s);
```

`document` 对象还有一个 `cookie` 属性，可以获取当前页面的Cookie。

Cookie是由服务器发送的key-value标示符。因为HTTP协议是无状态的，但是服务器要区分到底是哪个用户发过来的请求，就可以用Cookie来区分。当一个用户成功登录后，服务器发送一个Cookie给浏览器，例如 `user=ABC123XYZ(加密的字符串)...` ，此后，浏览器访问该网站时，会在请求头附上这个Cookie，服务器根据Cookie即可区分出用户。

Cookie还可以存储网站的一些设置，例如，页面显示的语言等等。

JavaScript可以通过 `document.cookie` 读取到当前页面的Cookie：

```
document.cookie; // 'v=123; remember=true; prefer=zh'
```

由于JavaScript能读取到页面的Cookie，而用户的登录信息通常也存在Cookie中，这就造成了巨大的安全隐患，这是因为在HTML页面中引入第三方的JavaScript代码是允许的：

```
<!-- 当前页面在wwwexample.com -->
<html>
    <head>
        <script src="http://www.foo.com/jquery.js"></script>
    </head>
    ...
</html>
```

如果引入的第三方的JavaScript中存在恶意代码，则 `www.foo.com` 网站将直接获取到 `www.example.com` 网站的用户登录信息。

为了解决这个问题，服务器在设置Cookie时可以使用 `httpOnly` ，设定了 `httpOnly` 的Cookie将不能被JavaScript读取。这个行为由浏览器实现，主流浏览器均支持 `httpOnly` 选项，IE从IE6 SP1开始支持。

为了确保安全，服务器端在设置Cookie时，应该始终坚持使用 `httpOnly` 。

history

`history` 对象保存了浏览器的历史记录，JavaScript可以调用 `history` 对象的 `back()` 或 `forward ()`，相当于用户点击了浏览器的“后退”或“前进”按钮。

这个对象属于历史遗留对象，对于现代Web页面来说，由于大量使用AJAX和页面交互，简单粗暴地调用 `history.back()` 可能会让用户感到非常愤怒。

新手开始设计Web页面时喜欢在登录页登录成功时调用 `history.back()`，试图回到登录前的页面。这是一种错误的方法。

任何情况，你都不应该使用 `history` 这个对象了。

操作DOM

由于HTML文档被浏览器解析后就是一棵DOM树，要改变HTML的结构，就需要通过JavaScript来操作DOM。

始终记住DOM是一个树形结构。操作一个DOM节点实际上就是这么几个操作：

- 更新：更新该DOM节点的内容，相当于更新了该DOM节点表示的HTML的内容；
- 遍历：遍历该DOM节点下的子节点，以便进行进一步操作；
- 添加：在该DOM节点下新增一个子节点，相当于动态增加了一个HTML节点；
- 删除：将该节点从HTML中删除，相当于删掉了该DOM节点的内容以及它包含的所有子节点。

在操作一个DOM节点前，我们需要通过各种方式先拿到这个DOM节点。最常用的方法是 `document.getElementById()` 和 `document.getElementsByTagName()` ，以及CSS选择器 `document.getElementsByClassName()` 。

由于ID在HTML文档中是唯一的，所以 `document.getElementById()` 可以直接定位唯一的一个DOM节点。 `document.getElementsByTagName()` 和 `document.getElementsByClassName()` 总是返回一组DOM节点。要精确地选择DOM，可以先定位父节点，再从父节点开始选择，以缩小范围。

例如：

```
// 返回ID为'test'的节点：
var test = document.getElementById('test');

// 先定位ID为'test-table'的节点，再返回其内部所有tr节点：
var trs = document.getElementById('test-table').getElementsByTagNar

// 先定位ID为'test-div'的节点，再返回其内部所有class包含red的节点：
var reds = document.getElementById('test-div').getElementsByClassNa

// 获取节点test下的所有直属子节点:
var cs = test.children;

// 获取节点test下第一个、最后一个子节点：
var first = test.firstElementChild;
var last = test.lastElementChild;
```

第二种方法是使用 `querySelector()` 和 `querySelectorAll()` ，需要了解selector语法，然后使用条件来获取节点，更加方便：

```
// 通过querySelector获取ID为q1的节点：
var q1 = document.querySelector('#q1');

// 通过querySelectorAll获取q1节点内的符合条件的所有节点：
var ps = q1.querySelectorAll('div.highlighted > p');
```

注意：低版本的IE<8不支持 `querySelector` 和 `querySelectorAll` 。IE8仅有限支持。

严格地讲，我们这里的DOM节点是指 `Element` ，但是DOM节点实际上是 `Node` ，在HTML中， `Node` 包括 `Element` 、 `Comment` 、 `CDATA_SECTION` 等很多种，以及根节点 `Document` 类型，但是，绝大多数时候我们只关心 `Element` ，也就是实际控制页面结构的 `Node` ，其他类型的 `Node` 忽略即可。根节点 `Document` 已经自动绑定为全局变量 `document` 。

练习

如下的HTML结构：

JavaScript

Java

Python

Ruby

Swift

Scheme

Haskell

```
<!-- HTML结构 -->
<div id="test-div">
<div class="c-red">
    <p id="test-p">JavaScript</p>
    <p>Java</p>
  </div>
  <div class="c-red c-green">
    <p>Python</p>
    <p>Ruby</p>
    <p>Swift</p>
  </div>
  <div class="c-green">
    <p>Scheme</p>
    <p>Haskell</p>
  </div>
</div>
```

请选择出指定条件的节点：

```
'use strict';

// 选择<p>JavaScript</p>:
var js = ???;

// 选择<p>Python</p>,<p>Ruby</p>,<p>Swift</p>:
var arr = ???;

// 选择<p>Haskell</p>:
var haskell = ???;

// 测试:
if (!js || js.innerText !== 'JavaScript') {
    alert('选择JavaScript失败!');
} else if (!arr || arr.length !== 3 || !arr[0] || !arr[1] || !arr[2
    alert('选择Python,Ruby,Swift失败!');
} else if (!haskell || haskell.innerText !== 'Haskell') {
    alert('选择Haskell失败!');
} else {
    alert('测试通过!');
}
```

更新DOM

拿到一个DOM节点后，我们可以对它进行更新。

可以直接修改节点的文本，方法有两种：

一种是修改 `innerHTML` 属性，这个方式非常强大，不但可以修改一个DOM节点的文本内容，还可以直接通过HTML片段修改DOM节点内部的子树：

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置文本为abc:
p.innerHTML = 'ABC'; // <p id="p-id">ABC</p>
// 设置HTML:
p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';
// <p>...</p>的内部结构已修改
```

用 `innerHTML` 时要注意，是否需要写入HTML。如果写入的字符串是通过网络拿到了，要注意对字符编码来避免XSS攻击。

第二种是修改 `innerText` 或 `textContent` 属性，这样可以自动对字符串进行HTML编码，保证无法设置任何HTML标签：

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置文本:
p.innerText = '<script>alert("Hi")</script>';
// HTML被自动编码，无法设置一个<script>节点:
// <p id="p-id">&lt;script&gt;alert("Hi")&lt;/script&gt;</p>
```

两者的区别在于读取属性时， `innerText` 不返回隐藏元素的文本，而 `textContent` 返回所有文本。另外注意IE<9不支持 `textContent` 。

修改CSS也是经常需要的操作。DOM节点的 `style` 属性对应所有的CSS，可以直接获取或设置。因为CSS允许 `font-size` 这样的名称，但它并非JavaScript有效的属性名，所以需要在JavaScript中改写为驼峰式命名 `fontSize` ：

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置CSS:
p.style.color = '#ff0000';
p.style.fontSize = '20px';
p.style.paddingTop = '2em';
```

练习

有如下的HTML结构：

javascript

Java

```
<!-- HTML结构 -->
<div id="test-div">
  <p id="test-js">javascript</p>
  <p>Java</p>
</div>
```

请尝试获取指定节点并修改：

```
'use strict';

// 获取<p>javascript</p>节点:
var js = ???;

// 修改文本为JavaScript:
// TODO:

// 修改CSS为: color: #ff0000, font-weight: bold
// TODO:

// 测试:
if (js && js.parentNode && js.parentNode.id === 'test-div' && js.id
    if (js.innerText === 'JavaScript') {
        if (js.style && js.style.fontWeight === 'bold' && (js.style
            alert('测试通过!');
        } else {
            alert('CSS样式测试失败!');
        }
    } else {
        alert('文本测试失败!');
    }
} else {
    alert('节点测试失败!');
}
```

插入DOM

当我们获得了某个DOM节点，想在这个DOM节点内插入新的DOM，应该如何做？

如果这个DOM节点是空的，例如，<div></div>，那么，直接使用 innerHTML = 'child' 就可以修改DOM节点的内容，相当于“插入”了新的DOM节点。

如果这个DOM节点不是空的，那就不能这么做，因为 innerHTML 会直接替换掉原来的所有子节点。

有两个办法可以插入新的节点。一个是使用 appendChild ，把一个子节点添加到父节点的最后一个子节点。例如：

```
<!-- HTML结构 -->
<p id="js">JavaScript</p>
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

把 <p id="js">JavaScript</p> 添加到 <div id="list"> 的最后一项：

```
var
    js = document.getElementById('js'),
    list = document.getElementById('list');
list.appendChild(js);
```

现在，HTML结构变成了这样：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
    <p id="js">JavaScript</p>
</div>
```

因为我们插入的 `js` 节点已经存在于当前的文档树，因此这个节点首先会从原先的位置删除，再插入到新的位置。

更多的时候我们会从零创建一个新的节点，然后插入到指定位置：

```
var
    list = document.getElementById('list'),
    haskell = document.createElement('p');
haskell.id = 'haskell';
haskell.innerText = 'Haskell';
list.appendChild(haskell);
```

这样我们就动态添加了一个新的节点：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
    <p id="haskell">Haskell</p>
</div>
```

动态创建一个节点然后添加到DOM树中，可以实现很多功能。举个例子，下面的代码动态创建了一个 `<style>` 节点，然后把它添加到 `<head>` 节点的末尾，这样就动态地给文档添加了新的CSS定义：

```
var d = document.createElement('style');
d.setAttribute('type', 'text/css');
d.innerHTML = 'p { color: red }';
document.getElementsByTagName('head')[0].appendChild(d);
```

可以在Chrome的控制台执行上述代码，观察页面样式的变化。

insertBefore

如果我们要把子节点插入到指定的位置怎么办？可以使用 `parentElement.insertBefore(newElement, referenceElement);` ，子节点会插入到 `referenceElement` 之前。

还是以上面的HTML为例，假定我们要把 `Haskell` 插入到 `Python` 之前：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

可以这么写：

```
var
    list = document.getElementById('list'),
    ref = document.getElementById('python'),
    haskell = document.createElement('p');
haskell.id = 'haskell';
haskell.innerText = 'Haskell';
list.insertBefore(haskell, ref);
```

新的HTML结构如下：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="haskell">Haskell</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

可见，使用 `insertBefore` 重点是要拿到一个“参考子节点”的引用。很多时候，需要循环一个父节点的所有子节点，可以通过迭代 `children` 属性实现：

```
var
    i, c,
    list = document.getElementById('list');
for (i = 0; i < list.children.length; i++) {
    c = list.children[i]; // 拿到第i个子节点
}
```

练习

对于一个已有的HTML结构：

1. Scheme
2. JavaScript
3. Python
4. Ruby
5. Haskell

```
<!-- HTML结构 -->
<ol id="test-list">
    <li class="lang">Scheme</li>
    <li class="lang">JavaScript</li>
    <li class="lang">Python</li>
    <li class="lang">Ruby</li>
    <li class="lang">Haskell</li>
</ol>
```

按字符串顺序重新排序DOM节点：

```
'use strict';

// sort list:

// 测试:
;(function () {
    var
        arr, i,
        t = document.getElementById('test-list');
    if (t && t.children && t.children.length === 5) {
        arr = [];
        for (i=0; i<t.children.length; i++) {
            arr.push(t.children[i].innerText);
        }
        if (arr.toString() === ['Haskell', 'JavaScript', 'Python',
            alert('测试通过!');
        }
        else {
            alert('测试失败: ' + arr.toString());
        }
    }
    else {
        alert('测试失败!');
    }
})();
```

删除DOM

删除一个DOM节点就比插入要容易得多。

要删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的 `removeChild` 把自己删掉：

```
// 拿到待删除节点:
var self = document.getElementById('to-be-removed');
// 拿到父节点:
var parent = self.parentElement;
// 删除:
var removed = parent.removeChild(self);
removed === self; // true
```

注意到删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置。

当你遍历一个父节点的子节点并进行删除操作时，要注意， `children` 属性是一个只读属性，并且它在子节点变化时会实时更新。

例如，对于如下HTML结构：

```
<div id="parent">
    <p>First</p>
    <p>Second</p>
</div>
```

当我们用如下代码删除子节点时：

```
var parent = document.getElementById('parent');
parent.removeChild(parent.children[0]);
parent.removeChild(parent.children[1]); // <-- 浏览器报错
```

浏览器报错：`parent.children[1]` 不是一个有效的节点。原因就在于，当 `<p>First</p>` 节点被删除后，`parent.children` 的节点数量已经从2变为了1，索引 `[1]` 已经不存在了。

因此，删除多个节点时，要注意 `children` 属性时刻都在变化。

练习

- JavaScript
- Swift
- HTML
- ANSI C
- CSS
- DirectX

```
<!-- HTML结构 -->
<ul id="test-list">
    <li>JavaScript</li>
    <li>Swift</li>
    <li>HTML</li>
    <li>ANSI C</li>
    <li>CSS</li>
    <li>DirectX</li>
</ul>
```

把与Web开发技术不相关的节点删掉：

```
'use strict';

// TODO

// 测试:
;(function () {
    var
        arr, i,
        t = document.getElementById('test-list');
    if (t && t.children && t.children.length === 3) {
        arr = [];
        for (i = 0; i < t.children.length; i ++) {
            arr.push(t.children[i].innerText);
        }
        if (arr.toString() === ['JavaScript', 'HTML', 'CSS'].toStri
            alert('测试通过!');
        }
        else {
            alert('测试失败: ' + arr.toString());
        }
    }
    else {
        alert('测试失败!');
    }
})();
```

操作表单

用JavaScript操作表单和操作DOM是类似的，因为表单本身也是DOM树。

不过表单的输入框、下拉框等可以接收用户输入，所以用JavaScript来操作表单，可以获得用户输入的内容，或者对一个输入框设置新的内容。

HTML表单的输入控件主要有以下几种：

- 文本框，对应的 `<input type="text">` ，用于输入文本；
- 口令框，对应的 `<input type="password">` ，用于输入口令；
- 单选框，对应的 `<input type="radio">` ，用于选择一项；
- 复选框，对应的 `<input type="checkbox">` ，用于选择多项；
- 下拉框，对应的 `<select>` ，用于选择一项；
- 隐藏文本，对应的 `<input type="hidden">` ，用户不可见，但表单提交时会把隐藏文本发送到服务器。

获取值

如果我们获得了一个 `<input>` 节点的引用，就可以直接调用 `value` 获得对应的用户输入值：

```
// <input type="text" id="email">
var input = document.getElementById('email');
input.value; // '用户输入的值'
```

这种方式可以应用于 `text` 、 `password` 、 `hidden` 以及 `select` 。但是，对于单选框和复选框， `value` 属性返回的永远是HTML预设的值，而我们需要获得的实际是用户是否“勾上了”选项，所以应该用 `checked` 判断：

```
// <label><input type="radio" name="weekday" id="monday" value="1">
// <label><input type="radio" name="weekday" id="tuesday" value="2"
var mon = document.getElementById('monday');
var tue = document.getElementById('tuesday');
mon.value; // '1'
tue.value; // '2'
mon.checked; // true或者false
tue.checked; // true或者false
```

设置值

设置值和获取值类似，对于 `text` 、 `password` 、 `hidden` 以及 `select` ，直接设置 `value` 就可以：

```
// <input type="text" id="email">
var input = document.getElementById('email');
input.value = 'test@example.com'; // 文本框的内容已更新
```

对于单选框和复选框，设置 `checked` 为 `true` 或 `false` 即可。

HTML5控件

HTML5新增了大量标准控件，常用的包括 `date` 、 `datetime` 、 `datetime-local` 、 `color` 等，它们都使用 <input> 标签：

```
<input type="date" value="2015-07-01">
```

```
<input type="datetime-local" value="2015-07-01T02:03:04">
```

```
<input type="color" value="#ff0000">
```

不支持HTML5的浏览器无法识别新的控件，会把它们当做 `type="text"` 来显示。支持HTML5的浏览器将获得格式化的字符串。例如， `type="date"` 类型的 `input` 的 `value` 将保证是一个有效的 `YYYY-MM-DD` 格式的日期，或者空字符串。

提交表单

最后，JavaScript可以以两种方式来处理表单的提交（AJAX方式在后面章节介绍）。

方式一是通过 `<form>` 元素的 `submit()` 方法提交一个表单，例如，响应一个 `<button>` 的 `click` 事件，在JavaScript代码中提交表单：

```
<!-- HTML -->
<form id="test-form">
    <input type="text" name="test">
    <button type="button" onclick="doSubmitForm()">Submit</button>
</form>

<script>
function doSubmitForm() {
    var form = document.getElementById('test-form');
    // 可以在此修改form的input...
    // 提交form:
    form.submit();
}
</script>
```

这种方式的缺点是扰乱了浏览器对form的正常提交。浏览器默认点击 `<button type="submit">` 时提交表单，或者用户在最后一个输入框按回车键。因此，第二种方式是响应 `<form>` 本身的 `onsubmit` 事件，在提交form时作修改：

```
<!-- HTML -->
<form id="test-form" onsubmit="return checkForm()">
    <input type="text" name="test">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
    var form = document.getElementById('test-form');
    // 可以在此修改form的input...
    // 继续下一步:
    return true;
}
</script>
```

注意要 `return true` 来告诉浏览器继续提交，如果 `return false` ，浏览器将不会继续提交form，这种情况通常对应用户输入有误，提示用户错误信息后终止提交form。

在检查和修改 `<input>` 时，要充分利用 `<input type="hidden">` 来传递数据。

例如，很多登录表单希望用户输入用户名和口令，但是，安全考虑，提交表单时不传输明文口令，而是口令的MD5。普通JavaScript开发人员会直接修改 `<input>` ：

```
<!-- HTML -->
<form id="login-form" method="post" onsubmit="return checkForm()">
    <input type="text" id="username" name="username">
    <input type="password" id="password" name="password">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
    var pwd = document.getElementById('password');
    // 把用户输入的明文变为MD5:
    pwd.value = toMD5(pwd.value);
    // 继续下一步:
    return true;
}
</script>
```

这个做法看上去没啥问题，但用户输入了口令提交时，口令框的显示会突然从几个 `*` 变成32个 `*` （因为MD5有32个字符）。

要想不改变用户的输入，可以利用 `<input type="hidden">` 实现：

```
<!-- HTML -->
<form id="login-form" method="post" onsubmit="return checkForm()">
    <input type="text" id="username" name="username">
    <input type="password" id="input-password">
    <input type="hidden" id="md5-password" name="password">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
    var input_pwd = document.getElementById('input-password');
    var md5_pwd = document.getElementById('md5-password');
    // 把用户输入的明文变为MD5:
    md5_pwd.value = toMD5(input_pwd.value);
    // 继续下一步:
    return true;
}
</script>
```

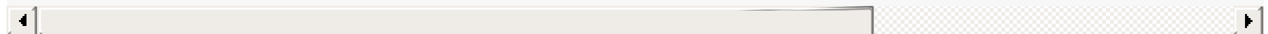
注意到 `id` 为 `md5-password` 的 `<input>` 标记了 `name="password"` ，而用户输入的 `id` 为 `input-password` 的 `<input>` 没有 `name` 属性。没有 `name` 属性的 `<input>` 的数据不会被提交。

练习

利用JavaScript检查用户注册信息是否正确，在以下情况不满足时报错并阻止提交表单：

- 用户名必须是3-10位英文字母或数字；
- 口令必须是6-20位；
- 两次输入口令必须一致。

```
<!-- HTML结构 -->
<form id="test-register" action="#" target="_blank" onsubmit="retur
    <p id="test-error" style="color:red"></p>
    <p>
        用户名: <input type="text" id="username" name="username">
    </p>
    <p>
        口令: <input type="password" id="password" name="password">
    </p>
    <p>
        重复口令: <input type="password" id="password-2">
    </p>
    <p>
        <button type="submit">提交</button> <button type="reset">重
    </p>
</form>
```



```
'use strict';
var checkRegisterForm = function () {

    // TODO:
    return false;
}

// 测试:
;(function () {
    window.testFormHandler = checkRegisterForm;
    var form = document.getElementById('test-register');
    if (form.dispatchEvent) {
        var event = new Event('submit', {
            bubbles: true,
            cancelable: true
          });
        form.dispatchEvent(event);
    } else {
        form.fireEvent('onsubmit');
    }
})();
```

操作文件

在HTML表单中，可以上传文件的唯一控件就是 `<input type="file">` 。

注意：当一个表单包含 `<input type="file">` 时，表单的 `enctype` 必须指定为 `multipart/form-data` ， `method` 必须指定为 `post` ，浏览器才能正确编码并以 `multipart/form-data` 格式发送表单的数据。

出于安全考虑，浏览器只允许用户点击 `<input type="file">` 来选择本地文件，用JavaScript对 `<input type="file">` 的 `value` 赋值是没有任何效果的。当用户选择了上传某个文件后，JavaScript也无法获得该文件的真实路径：

通常，上传的文件都由后台服务器处理，JavaScript可以在提交表单时对文件扩展名做检查，以便防止用户上传无效格式的文件：

```
var f = document.getElementById('test-file-upload');
var filename = f.value; // 'C:\fakepath\test.png'
if (!filename || !(filename.endsWith('.jpg') || filename.endsWith(
    alert('Can only upload image file.');
    return false;
}
```

File API

由于JavaScript对用户上传的文件操作非常有限，尤其是无法读取文件内容，使得很多需要操作文件的网页不得不用Flash这样的第三方插件来实现。

随着HTML5的普及，新增的File API允许JavaScript读取文件内容，获得更多的文件信息。

HTML5的File API提供了 `File` 和 `FileReader` 两个主要对象，可以获得文件信息并读取文件。

下面的例子演示了如何读取用户选取的图片文件，并在一个 `<div>` 中预览图像：

```
var
    fileInput = document.getElementById('test-image-file'),
    info = document.getElementById('test-file-info'),
    preview = document.getElementById('test-image-preview');
// 监听change事件:
fileInput.addEventListener('change', function () {
    // 清除背景图片:
    preview.style.backgroundImage = '';
    // 检查文件是否选择:
    if (!fileInput.value) {
        info.innerHTML = '没有选择文件';
        return;
    }
    // 获取File引用:
    var file = fileInput.files[0];
    // 获取File信息:
    info.innerHTML = '文件: ' + file.name + '<br>' +
                     '大小: ' + file.size + '<br>' +
                     '修改: ' + file.lastModifiedDate;
    if (file.type !== 'image/jpeg' && file.type !== 'image/png' &&
        alert('不是有效的图片文件!');
        return;
    }
    // 读取文件:
    var reader = new FileReader();
    reader.onload = function(e) {
        var
            data = e.target.result; // 'data:image/jpeg;base64,/9j/
        preview.style.backgroundImage = 'url(' + data + ')';
    };
    // 以DataURL的形式读取文件:
    reader.readAsDataURL(file);
});
```

上面的代码演示了如何通过HTML5的File API读取文件内容。以DataURL的形式读取到的文件是一个字符串，类似于 `data:image/jpeg;base64,/9j/4AAQSk...(base64编码)...`，常用于设置图像。如果需要服务器端处理，把字符

串 `base64,` 后面的字符发送给服务器并用Base64解码就可以得到原始文件的二进制内容。

回调

上面的代码还演示了JavaScript的一个重要的特性就是单线程执行模式。在JavaScript中，浏览器的JavaScript执行引擎在执行JavaScript代码时，总是以单线程模式执行，也就是说，任何时候，JavaScript代码都不可能同时有多于1个线程在执行。

你可能会问，单线程模式执行的JavaScript，如何处理多任务？

在JavaScript中，执行多任务实际上都是异步调用，比如上面的代码：

```
reader.readAsDataURL(file);
```

就会发起一个异步操作来读取文件内容。因为是异步操作，所以我们在JavaScript代码中就不知道什么时候操作结束，因此需要先设置一个回调函数：

```
reader.onload = function(e) {
    // 当文件读取完成后，自动调用此函数:
};
```

当文件读取完成后，JavaScript引擎将自动调用我们设置的回调函数。执行回调函数时，文件已经读取完毕，所以我们可以在回调函数内部安全地获得文件内容。

AJAX

AJAX不是JavaScript的规范，它只是一个哥们“发明”的缩写：Asynchronous JavaScript and XML，意思就是用JavaScript执行异步网络请求。

如果仔细观察一个Form的提交，你就会发现，一旦用户点击“Submit”按钮，表单开始提交，浏览器就会刷新页面，然后在新页面里告诉你操作是成功了还是失败了。如果不幸由于网络太慢或者其他原因，就会得到一个404页面。

这就是Web的运作原理：一次HTTP请求对应一个页面。

如果要让用户留在当前页面中，同时发出新的HTTP请求，就必须用JavaScript发送这个新请求，接收到数据后，再用JavaScript更新页面，这样一来，用户就感觉自己仍然停留在当前页面，但是数据却可以不断地更新。

最早大规模使用AJAX的就是Gmail，Gmail的页面在首次加载后，剩下的所有数据都依赖于AJAX来更新。

用JavaScript写一个完整的AJAX代码并不复杂，但是需要注意：AJAX请求是异步执行的，也就是说，要通过回调函数获得响应。

在现代浏览器上写AJAX主要依靠 `XMLHttpRequest` 对象：

```
'use strict';

function success(text) {
    var textarea = document.getElementById('test-response-text');
    textarea.value = text;
}

function fail(code) {
    var textarea = document.getElementById('test-response-text');
    textarea.value = 'Error code: ' + code;
}

var request = new XMLHttpRequest(); // 新建XMLHttpRequest对象

request.onreadystatechange = function () { // 状态发生变化时，函数被回
    if (request.readyState === 4) { // 成功完成
        // 判断响应结果:
        if (request.status === 200) {
            // 成功，通过responseText拿到响应的文本:
            return success(request.responseText);
        } else {
            // 失败，根据响应码判断失败原因:
            return fail(request.status);
        }
    } else {
        // HTTP请求还在继续...
    }
}

// 发送请求:
request.open('GET', '/api/categories');
request.send();

alert('请求已发送，请等待响应...');
```

对于低版本的IE，需要换一个 `ActiveXObject` 对象：

```
'use strict';

function success(text) {
    var textarea = document.getElementById('test-ie-response-text'
    textarea.value = text;
}

function fail(code) {
    var textarea = document.getElementById('test-ie-response-text'
    textarea.value = 'Error code: ' + code;
}

var request = new ActiveXObject('Microsoft.XMLHTTP'); // 新建Micros

request.onreadystatechange = function () { // 状态发生变化时，函数被回
    if (request.readyState === 4) { // 成功完成
        // 判断响应结果:
        if (request.status === 200) {
            // 成功，通过responseText拿到响应的文本:
            return success(request.responseText);
        } else {
            // 失败，根据响应码判断失败原因:
            return fail(request.status);
        }
    } else {
        // HTTP请求还在继续...
    }
}

// 发送请求:
request.open('GET', '/api/categories');
request.send();

alert('请求已发送，请等待响应...');
```

如果你想把标准写法和IE写法混在一起，可以这么写：

```
var request;
if (window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else {
    request = new ActiveXObject('Microsoft.XMLHTTP');
}
```

通过检测 `window` 对象是否有 `XMLHttpRequest` 属性来确定浏览器是否支持标准的 `XMLHttpRequest` 。注意，不要根据浏览器的 `navigator.userAgent` 来检测浏览器是否支持某个JavaScript特性，一是因为这个字符串本身可以伪造，二是通过IE版本判断JavaScript特性将非常复杂。

当创建了 `XMLHttpRequest` 对象后，要先设置 `onreadystatechange` 的回调函数。在回调函数中，通常我们只需通过 `readyState === 4` 判断请求是否完成，如果已完成，再根据 `status === 200` 判断是否是一个成功的响应。

`XMLHttpRequest` 对象的 `open()` 方法有3个参数，第一个参数指定是 `GET` 还是 `POST` ，第二个参数指定URL地址，第三个参数指定是否使用异步，默认是 `true` ，所以不用写。

注意，千万不要把第三个参数指定为 `false` ，否则浏览器将停止响应，直到AJAX请求完成。如果这个请求耗时10秒，那么10秒内你会发现浏览器处于“假死”状态。

最后调用 `send()` 方法才真正发送请求。 `GET` 请求不需要参数， `POST` 请求需要把body部分以字符串或者 `FormData` 对象传进去。

安全限制

上面代码的URL使用的是相对路径。如果你把它改为 `'http://www.sina.com.cn/'` ，再运行，肯定报错。在Chrome的控制台里，还可以看到错误信息。

这是因为浏览器的同源策略导致的。默认情况下，JavaScript在发送AJAX请求时，URL的域名必须和当前页面完全一致。

完全一致的意思是，域名要相同（ `www.example.com` 和 `example.com` 不同），协议要相同（ `http` 和 `https` 不同），端口号要相同（默认是 `:80` 端口，它和 `:8080` 就不同）。有的浏览器口子松一点，允许端口不同，大多数浏览器都会

严格遵守这个限制。

那是不是用JavaScript无法请求外域（就是其他网站）的URL了呢？方法还是有的，大概有这么几种：

一是通过Flash插件发送HTTP请求，这种方式可以绕过浏览器的安全限制，但必须安装Flash，并且跟Flash交互。不过Flash用起来麻烦，而且现在用得也越来越少了。

二是通过在同源域名下架设一个代理服务器来转发，JavaScript负责把请求发送到代理服务器：

```
'/proxy?url=http://www.sina.com.cn'
```

代理服务器再把结果返回，这样就遵守了浏览器的同源策略。这种方式麻烦之处在于需要服务器端额外做开发。

第三种方式称为JSONP，它有个限制，只能用GET请求，并且要求返回JavaScript。这种方式跨域实际上是利用了浏览器允许跨域引用JavaScript资源：

```
<html>
<head>
    <script src="http://example.com/abc.js"></script>
    ...
</head>
<body>
...
</body>
</html>
```

JSONP通常以函数调用的形式返回，例如，返回JavaScript内容如下：

```
foo('data');
```

这样一来，我们如果在页面中先准备好 `foo()` 函数，然后给页面动态加一个 `<script>` 节点，相当于动态读取外域的JavaScript资源，最后就等着接收回调了。

以163的股票查询URL为例，对于 URL：http://api.money.126.net/data/feed/0000001,1399001?callback=refreshPrice，你将得到如下返回：

```
refreshPrice({"0000001":{"code": "0000001", ... });
```

因此我们需要首先在页面中准备好回调函数：

```
function refreshPrice(data) {
    var p = document.getElementById('test-jsonp');
    p.innerHTML = '当前价格：' +
        data['0000001'].name +': ' +
        data['0000001'].price + '；' +
        data['1399001'].name + ': ' +
        data['1399001'].price;
}
```

当前价格：

<button type="button" onclick="getPrice()">刷新</button>

最后用 `getPrice()` 函数触发：

```
function getPrice() {
    var
        js = document.createElement('script'),
        head = document.getElementsByTagName('head')[0];
    js.src = 'http://api.money.126.net/data/feed/0000001,1399001?ca
    head.appendChild(js);
}
```

就完成了跨域加载数据。

CORS

如果浏览器支持HTML5，那么就可以一劳永逸地使用新的跨域策略：CORS了。

CORS全称Cross-Origin Resource Sharing，是HTML5规范定义的如何跨域访问资源。

了解CORS前，我们先搞明白概念：

Origin表示本域，也就是浏览器当前页面的域。当JavaScript向外域（如sina.com）发起请求后，浏览器收到响应后，首先检查 `Access-Control-Allow-Origin` 是否包含本域，如果是，则此次跨域请求成功，如果不是，则请求失败，JavaScript将无法获取到响应的任何数据。

用一个图来表示就是：



假设本域是 `my.com` ，外域是 `sina.com` ，只要响应头 `Access-Control-Allow-Origin` 为 `http://my.com` ，或者是 `*` ，本次请求就可以成功。

可见，跨域能否成功，取决于对方服务器是否愿意给你设置一个正确的 `Access-Control-Allow-Origin` ，决定权始终在对方手中。

上面这种跨域请求，称之为“简单请求”。简单请求包括GET、HEAD和POST（POST的Content-Type类型仅限 `application/x-www-form-urlencoded` 、 `multipart/form-data` 和 `text/plain` ），并且不能出现任何自定义头（例如， `X-Custom: 12345` ），通常能满足90%的需求。

无论你是否需要用JavaScript通过CORS跨域请求资源，你都要了解CORS的原理。最新的浏览器全面支持HTML5。在引用外域资源时，除了JavaScript和CSS外，都要验证CORS。例如，当你引用了某个第三方CDN上的字体文件时：

```
/* CSS */
@font-face {
  font-family: 'FontAwesome';
  src: url('http://cdn.com/fonts/fontawesome.ttf') format('truetype
}
```

如果该CDN服务商未正确设置 `Access-Control-Allow-Origin` ，那么浏览器无法加载字体资源。

对于PUT、DELETE以及其他类型如 `application/json` 的POST请求，在发送AJAX请求之前，浏览器会先发送一个 `OPTIONS` 请求（称为preflighted请求）到这个URL上，询问目标服务器是否接受：

```
OPTIONS /path/to/resource HTTP/1.1
Host: bar.com
Origin: http://my.com
Access-Control-Request-Method: POST
```

服务器必须响应并明确指出允许的Method：

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://my.com
Access-Control-Allow-Methods: POST, GET, PUT, OPTIONS
Access-Control-Max-Age: 86400
```

浏览器确认服务器响应的 `Access-Control-Allow-Methods` 头确实包含将要发送的AJAX请求的Method，才会继续发送AJAX，否则，抛出一个错误。

由于以 `POST` 、 `PUT` 方式传送JSON格式的数据在REST中很常见，所以要跨域正确处理 `POST` 和 `PUT` 请求，服务器端必须正确响应 `OPTIONS` 请求。

需要深入了解CORS的童鞋请移步W3C文档。

Promise

在JavaScript的世界中，所有代码都是单线程执行的。

由于这个“缺陷”，导致JavaScript的所有网络操作，浏览器事件，都必须是异步执行。异步执行可以用回调函数实现：

```
function callback() {
    console.log('Done');
}
console.log('before setTimeout()');
setTimeout(callback, 1000); // 1秒钟后调用callback函数
console.log('after setTimeout()');
```

观察上述代码执行，在Chrome的控制台输出可以看到：

```
before setTimeout()
after setTimeout()
(等待1秒后)
Done
```

可见，异步操作会在将来的某个时间点触发一个函数调用。

AJAX就是典型的异步操作。以上一节的代码为例：

```
request.onreadystatechange = function () {
    if (request.readyState === 4) {
        if (request.status === 200) {
            return success(request.responseText);
        } else {
            return fail(request.status);
        }
    }
}
```

把回调函数 `success(request.responseText)` 和 `fail(request.status)` 写到一个AJAX操作里很正常，但是不好看，而且不利于代码复用。

有没有更好的写法？比如写成这样：

```
var ajax = ajaxGet('http://...');
ajax.ifSuccess(success)
    .ifFail(fail);
```

这种链式写法的好处在于，先统一执行AJAX逻辑，不关心如何处理结果，然后，根据结果是成功还是失败，在将来的某个时候调用 `success` 函数或 `fail` 函数。

古人云："君子一诺千金"，这种"承诺将来会执行"的对象在JavaScript中称为Promise对象。

Promise有各种开源实现，在ES6中被统一规范，由浏览器直接支持。先测试一下你的浏览器是否支持Promise：

```
'use strict';

new Promise(function () {});

// 直接运行测试:
alert('支持Promise!');
```

我们先看一个最简单的Promise例子：生成一个0-2之间的随机数，如果小于1，则等待一段时间后返回成功，否则返回失败：

```
function test(resolve, reject) {
    var timeOut = Math.random() * 2;
    log('set timeout to: ' + timeOut + ' seconds.');
    setTimeout(function () {
        if (timeOut < 1) {
            log('call resolve()...');
            resolve('200 OK');
        }
        else {
            log('call reject()...');
            reject('timeout in ' + timeOut + ' seconds.');
        }
    }, timeOut * 1000);
}
```

这个 `test()` 函数有两个参数，这两个参数都是函数，如果执行成功，我们将调用 `resolve('200 OK')`，如果执行失败，我们将调用 `reject('timeout in ' + timeOut + ' seconds.')`。可以看出，`test()` 函数只关心自身的逻辑，并不关心具体的 `resolve` 和 `reject` 将如何处理结果。

有了执行函数，我们就可以用一个Promise对象来执行它，并在将来某个时刻获得成功或失败的结果：

```
var p1 = new Promise(test);
var p2 = p1.then(function (result) {
    console.log('成功：' + result);
});
var p3 = p2.catch(function (reason) {
    console.log('失败：' + reason);
});
```

变量 `p1` 是一个Promise对象，它负责执行 `test` 函数。由于 `test` 函数在内部是异步执行的，当 `test` 函数执行成功时，我们告诉Promise对象：

```
// 如果成功，执行这个函数：
p1.then(function (result) {
    console.log('成功：' + result);
});
```

当 `test` 函数执行失败时，我们告诉Promise对象：

```
p2.catch(function (reason) {
    console.log('失败：' + reason);
});
```

Promise对象可以串联起来，所以上述代码可以简化为：

```
new Promise(test).then(function (result) {
    console.log('成功：' + result);
}).catch(function (reason) {
    console.log('失败：' + reason);
});
```

实际测试一下，看看Promise是如何异步执行的：

```
'use strict';

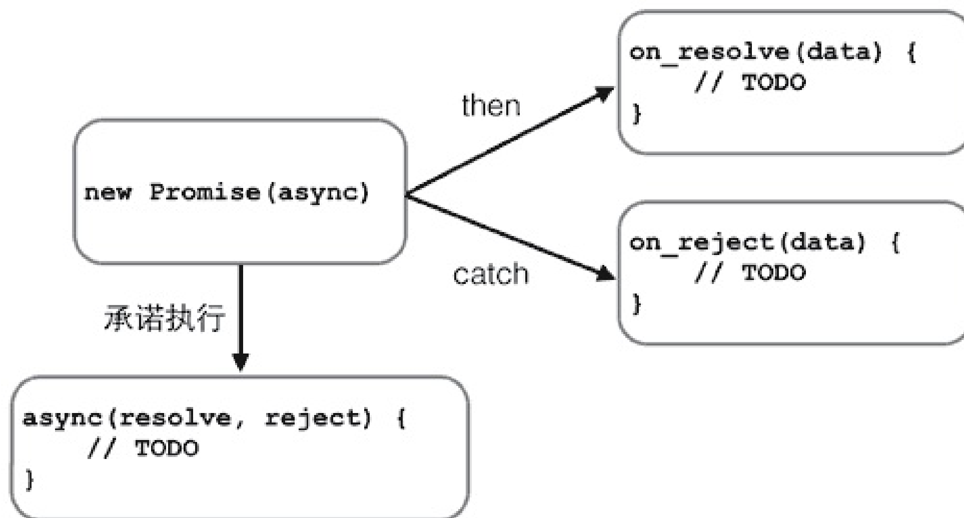
// 清除log:
var logging = document.getElementById('test-promise-log');
while (logging.children.length > 1) {
    logging.removeChild(logging.children[logging.children.length -
}

// 输出log到页面:
function log(s) {
    var p = document.createElement('p');
    p.innerHTML = s;
    logging.appendChild(p);
}


new Promise(function (resolve, reject) {
    log('start new Promise...');
    var timeOut = Math.random() * 2;
    log('set timeout to: ' + timeOut + ' seconds.');
    setTimeout(function () {
        if (timeOut < 1) {
            log('call resolve()...');
            resolve('200 OK');
        }
        else {
            log('call reject()...');
            reject('timeout in ' + timeOut + ' seconds.');
        }
    }, timeOut * 1000);
}).then(function (r) {
    log('Done: ' + r);
}).catch(function (reason) {
    log('Failed: ' + reason);
});
```

Log:

可见Promise最大的好处是在异步执行的流程中，把执行代码和处理结果的代码清晰地分离了：



Promise还可以做更多的事情，比如，有若干个异步任务，需要先做任务1，如果成功后再做任务2，任何任务失败则不再继续并执行错误处理函数。

要串行执行这样的异步任务，不用Promise需要写一层一层的嵌套代码。有了Promise，我们只需要简单地写：

```
job1.then(job2).then(job3).catch(handleError);
```

其中，`job1`、`job2` 和 `job3` 都是Promise对象。

下面的例子演示了如何串行执行一系列需要异步计算获得结果的任务：

```
'use strict';

var logging = document.getElementById('test-promise2-log');
while (logging.children.length > 1) {
    logging.removeChild(logging.children[logging.children.length -
}

function log(s) {
    var p = document.createElement('p');
    p.innerHTML = s;
    logging.appendChild(p);
}
```

```
// 0.5秒后返回input*input的计算结果:
function multiply(input) {
    return new Promise(function (resolve, reject) {
        log('calculating ' + input + ' x ' + input + '...');
        setTimeout(resolve, 500, input * input);
    });
}

// 0.5秒后返回input+input的计算结果:
function add(input) {
    return new Promise(function (resolve, reject) {
        log('calculating ' + input + ' + ' + input + '...');
        setTimeout(resolve, 500, input + input);
    });
}

var p = new Promise(function (resolve, reject) {
    log('start new Promise...');
    resolve(123);
});

p.then(multiply)
 .then(add)
 .then(multiply)
 .then(add)
 .then(function (result) {
    log('Got value: ' + result);
});
```

Log:

`setTimeout` 可以看成一个模拟网络等异步执行的函数。现在，我们把上一节的AJAX异步执行函数转换为Promise对象，看看用Promise如何简化异步处理：

```
'use strict';

// ajax函数将返回Promise对象:
function ajax(method, url, data) {
    var request = new XMLHttpRequest();
    return new Promise(function (resolve, reject) {
        request.onreadystatechange = function () {
            if (request.readyState === 4) {
                if (request.status === 200) {
                    resolve(request.responseText);
                } else {
                    reject(request.status);
                }
            }
        };
        request.open(method, url);
        request.send(data);
    });
}

var log = document.getElementById('test-promise-ajax-result');
var p = ajax('GET', '/api/categories');
p.then(function (text) { // 如果AJAX成功，获得响应内容
    log.innerText = text;
}).catch(function (status) { // 如果AJAX失败，获得响应代码
    log.innerText = 'ERROR: ' + status;
});
```

Result:

除了串行执行若干异步任务外，Promise还可以并行执行异步任务。

试想一个页面聊天系统，我们需要从两个不同的URL分别获得用户的个人信息和好友列表，这两个任务是可以并行执行的，用 `Promise.all()` 实现如下：

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 500, 'P1');
});
var p2 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 600, 'P2');
});
// 同时执行p1和p2，并在它们都完成后执行then:
Promise.all([p1, p2]).then(function (results) {
    console.log(results); // 获得一个Array: ['P1', 'P2']
});
```

有些时候，多个异步任务是为了容错。比如，同时向两个URL读取用户的个人信息，只需要获得先返回的结果即可。这种情况下，用 `Promise.race()` 实现：

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 500, 'P1');
});
var p2 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 600, 'P2');
});
Promise.race([p1, p2]).then(function (result) {
    console.log(result); // 'P1'
});
```

由于 `p1` 执行较快，Promise的 `then()` 将获得结果 `'P1'` 。 `p2` 仍在继续执行，但执行结果将被丢弃。

如果我们组合使用Promise，就可以把很多异步任务以并行和串行的方式组合起来执行。

Canvas

Canvas是HTML5新增的组件，它就像一块幕布，可以用JavaScript在上面绘制各种图表、动画等。

没有Canvas的年代，绘图只能借助Flash插件实现，页面不得不用JavaScript和Flash进行交互。有了Canvas，我们就再也不需要Flash了，直接使用JavaScript完成绘制。

一个Canvas定义了一个指定尺寸的矩形框，在这个范围内我们可以随意绘制：

```
<canvas id="test-canvas" width="300" height="200"></canvas>
```

由于浏览器对HTML5标准支持不一致，所以，通常在 `<canvas>` 内部添加一些说明性HTML代码，如果浏览器支持Canvas，它将忽略 `<canvas>` 内部的HTML，如果浏览器不支持Canvas，它将显示 `<canvas>` 内部的HTML：

```
<canvas id="test-stock" width="300" height="200">
    <p>Current Price: 25.51</p>
</canvas>
```

在使用Canvas前，用 `canvas.getContext` 来测试浏览器是否支持Canvas：

```
<!-- HTML代码 -->
<canvas id="test-canvas" width="200" heigth="100">
    <p>你的浏览器不支持Canvas</p>
</canvas>
```

```
'use strict';

var canvas = document.getElementById('test-canvas');
if (canvas.getContext) {
    alert('你的浏览器支持Canvas!');
} else {
    alert('你的浏览器不支持Canvas!');
}
```

`getContext('2d')` 方法让我们拿到一个 `CanvasRenderingContext2D` 对象，所有的绘图操作都需要通过这个对象完成。

```
var ctx = canvas.getContext('2d');
```

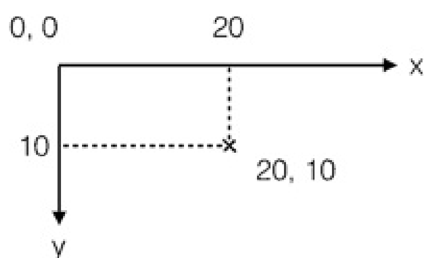
如果需要绘制3D怎么办？HTML5还有一个WebGL规范，允许在Canvas中绘制3D图形：

```
gl = canvas.getContext("webgl");
```

本节我们只专注于绘制2D图形。

绘制形状

我们可以在Canvas上绘制各种形状。在绘制前，我们需要先了解一下Canvas的坐标系统：



Canvas的坐标以左上角为原点，水平向右为X轴，垂直向下为Y轴，以像素为单位，所以每个点都是非负整数。

`CanvasRenderingContext2D` 对象有若干方法来绘制图形：

```
'use strict';

var
    canvas = document.getElementById('test-shape-canvas'),
    ctx = canvas.getContext('2d');

ctx.clearRect(0, 0, 200, 200); // 擦除(0,0)位置大小为200x200的矩形，擦
ctx.fillStyle = '#dddddd'; // 设置颜色
ctx.fillRect(10, 10, 130, 130); // 把(10,10)位置大小为130x130的矩形涂色
// 利用Path绘制复杂路径:
var path=new Path2D();
path.arc(75, 75, 50, 0, Math.PI*2, true);
path.moveTo(110,75);
path.arc(75, 75, 35, 0, Math.PI, false);
path.moveTo(65, 65);
path.arc(60, 65, 5, 0, Math.PI*2, true);
path.moveTo(95, 65);
path.arc(90, 65, 5, 0, Math.PI*2, true);
ctx.strokeStyle = '#0000ff';
ctx.stroke(path);
```

绘制文本

绘制文本就是在指定的位置输出文本，可以设置文本的字体、样式、阴影等，与CSS完全一致：

```
'use strict';

var
    canvas = document.getElementById('test-text-canvas'),
    ctx = canvas.getContext('2d');

ctx.clearRect(0, 0, canvas.width, canvas.height);
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 2;
ctx.shadowColor = '#666666';
ctx.font = '24px Arial';
ctx.fillStyle = '#333333';
ctx.fillText('带阴影的文字', 20, 40);
```

Canvas除了能绘制基本的形状和文本，还可以实现动画、缩放、各种滤镜和像素转换等高级操作。如果要实现非常复杂的操作，考虑以下优化方案：

- 通过创建一个不可见的Canvas来绘图，然后将最终绘制结果复制到页面的可见Canvas中；
- 尽量使用整数坐标而不是浮点数；
- 可以创建多个重叠的Canvas绘制不同的层，而不是在一个Canvas中绘制非常复杂的图；
- 背景图片如果不变可以直接用 `<img>` 标签并放到最底层。

练习

请根据从163获取的JSON数据绘制最近30个交易日的K线图，数据已处理为包含一组对象的数组：

```
window.drawStock = function (data) {
    var
        canvas = document.getElementById('stock-canvas'),
        MAX_X = canvas.width,
        MAX_Y = canvas.height,
        ctx = canvas.getContext('2d');
```

```
    var low = data.reduce(function (prev, x) {
        return x.low < prev.low ? x : prev;
    });
    var high = data.reduce(function (prev, x) {
        return x.high > prev.high ? x : prev;
    });

    var chg = high.high - low.low;

    // index range:
    var lowest = Math.floor(low.low - chg * 0.1);
    var highest = Math.floor(high.high + chg * 0.1 + 1);

    var calcY = function (idx) {
        return MAX_Y * (highest - idx) / (highest - lowest);
    };

    var drawAtX = function (x, k) {
        var
            tmp,
            y1 = calcY(k.open),
            y2 = calcY(k.close);
        if (y1 > y2) {
            tmp = y1;
            y1 = y2;
            y2 = tmp;
        }
        ctx.fillStyle = (k.open > k.close) ? '#00ff00' : '#ff0000'
        ctx.fillRect(x, calcY(k.high), 1, calcY(k.low) - calcY(k.h
        ctx.fillRect(x-2, y1, 5, y2 - y1);
    };

    ctx.clearRect(0, 0, MAX_X, MAX_Y);

    ctx.font = '12px serif';
    ctx.textAlign = 'right';
    ctx.fillStyle = '#000000';
    ctx.fillText(String(Math.floor(high.high)), 40, 15);
    ctx.fillText(String(Math.floor(low.low)), 40, MAX_Y - 20);
```

```
    var i, x;
    for (i=0; i<data.length; i++) {
        x = i * 8 + 50;
        drawAtX(x, data[i]);
    }
};
```

```
'use strict';

window.loadStockData = function (r) {
    var
        NUMS = 30,
        data = r.data;
    if (data.length > NUMS) {
        data = data.slice(data.length - NUMS);
    }
    data = data.map(function (x) {
        return {
            date: x[0],
            open: x[1],
            close: x[2],
            high: x[3],
            low: x[4],
            vol: x[5],
            change: x[6]
        };
    });
    window.drawStock(data);
}

window.drawStock = function (data) {

    var
        canvas = document.getElementById('stock-canvas'),
        width = canvas.width,
        height = canvas.height,
        ctx = canvas.getContext('2d');
```



```
    console.log(JSON.stringify(data[0])); // {"date":"20150602","o
    ctx.clearRect(0, 0, width, height);
    ctx.fillText('Test Canvas', 10, 10);

};

// 加载最近30个交易日的K线图数据:
var js = document.createElement('script');
js.src = 'http://img1.money.126.net/data/hs/kline/day/history/2015
document.getElementsByTagName('head')[0].appendChild(js);
```

下载为图片

jQuery

你可能听说过jQuery，它名字起得很土，但却是JavaScript世界中使用最广泛的一个库。

江湖传言，全世界大约有80~90%的网站直接或间接地使用了jQuery。鉴于它如此流行，又如此好用，所以每一个入门JavaScript的前端工程师都应该了解和学习它。

jQuery这么流行，肯定是因为它解决了一些很重要的问题。实际上，jQuery能帮我们干这些事情：

- 消除浏览器差异：你不需要自己写冗长的代码来针对不同的浏览器来绑定事件，编写AJAX等代码；
- 简洁的操作DOM的方法：写 `$('#test')` 肯定比 `document.getElementById('test')` 来得简洁；
- 轻松实现动画、修改CSS等各种操作。

jQuery的理念“Write Less, Do More“，让你写更少的代码，完成更多的工作！

jQuery版本

目前jQuery有1.x和2.x两个主要版本，区别在于2.x移除了对古老的IE 6、7、8的支持，因此2.x的代码更精简。选择哪个版本主要取决于你是否想支持IE 6~8。

从jQuery官网可以下载最新版本。jQuery只是一个 `jquery-xxx.js` 文件，但你会看到有compressed（已压缩）和uncompressed（未压缩）两种版本，使用时完全一样，但如果你想深入研究jQuery源码，那就用uncompressed版本。

使用jQuery

使用jQuery只需要在页面的 `<head>` 引入jQuery文件即可：

```
<html>
<head>
    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    ...
</head>
<body>
    ...
</body>
</html>
```

好消息是，当你在学习这个教程时，由于网站本身已经引用了jQuery，所以你可以直接使用：

```
'use strict';

alert('jQuery版本：' + $.fn.jquery);
```

$符号

`$` 是著名的jQuery符号。实际上，jQuery把所有功能全部封装在一个全局变量 `jQuery` 中，而 `$` 也是一个合法的变量名，它是变量 `jQuery` 的别名：

```
window.jQuery; // jQuery(selector, context)
window.$; // jQuery(selector, context)
$ === jQuery; // true
typeof($); // 'function'
```

`$` 本质上就是一个函数，但是函数也是对象，于是 `$` 除了可以直接调用外，也可以有很多其他属性。

注意，你看到的 `$` 函数名可能不是 `jQuery(selector, context)` ，因为很多JavaScript压缩工具可以对函数名和参数改名，所以压缩过的jQuery源码 `$` 函数可能变成 `a(b, c)` 。

绝大多数时候，我们都直接用 `$` （因为写起来更简单嘛）。但是，如果 `$` 这个变量不幸地被占用了，而且还不能改，那我们就只能让 `jQuery` 把 `$` 变量交出来，然后就只能使用 `jQuery` 这个变量：

```
$; // jQuery(selector, context)
jQuery.noConflict();
$; // undefined
jQuery; // jQuery(selector, context)
```

这种黑魔法的原理是jQuery在占用 `$` 之前，先在内部保存了原来的 `$` ,调用 `jQuery.noConflict()` 时会把原来保存的变量还原。

选择器

选择器是jQuery的核心。一个选择器写出来类似 `$('#dom-id')` 。

为什么jQuery要发明选择器？回顾一下DOM操作中我们经常使用的代码：

```
// 按ID查找：
var a = document.getElementById('dom-id');

// 按tag查找：
var divs = document.getElementsByTagNames('div');

// 查找<p class="red">：
var ps = document.getElementsByTagNames('p');
// 过滤出class="red":
// TODO:

// 查找<table class="green">里面的所有<tr>：
var table = ...
for (var i=0; i<table.children; i++) {
    // TODO: 过滤出<tr>
}
```

这些代码实在太繁琐了，并且，在层级关系中，例如，查找 `<table class="green">` 里面的所有 `<tr>` ，一层循环实际上是错的，因为 `<table>` 的标准写法是：

```
<table>
    <tbody>
        <tr>...</tr>
        <tr>...</tr>
    </tbody>
</table>
```

很多时候，需要递归查找所有子节点。

jQuery的选择器就是帮助我们快速定位到一个或多个DOM节点。

按ID查找

如果某个DOM节点有 `id` 属性，利用jQuery查找如下：

```
// 查找<div id="abc">:
var div = $('#abc');
```

注意， `#abc` 以 `#` 开头。返回的对象是jQuery对象。

什么是jQuery对象？jQuery对象类似数组，它的每个元素都是一个引用了DOM节点的对象。

以上面的查找为例，如果 `id` 为 `abc` 的 `<div>` 存在，返回的jQuery对象如下：

```
[<div id="abc">...</div>]
```

如果 `id` 为 `abc` 的 `<div>` 不存在，返回的jQuery对象如下：

```
[]
```

总之jQuery的选择器不会返回 `undefined` 或者 `null` ，这样的好处是你不必在下一行判断 `if (div === undefined)` 。

jQuery对象和DOM对象之间可以互相转化：

```
var div = $('#abc'); // jQuery对象
var divDom = div.get(0); // 假设存在div，获取第1个DOM元素
var another = $(divDom); // 重新把DOM包装为jQuery对象
```

通常情况下你不需要获取DOM对象，直接使用jQuery对象更加方便。如果你拿到了一个DOM对象，那可以简单地调用 `$(aDomObject)` 把它变成jQuery对象，这样就可以方便地使用jQuery的API了。

按tag查找

按tag查找只需要写上tag名称就可以了：

```
var ps = $('p'); // 返回所有<p>节点
ps.length; // 数一数页面有多少个<p>节点
```

按class查找

按class查找注意在class名称前加一个 `.` ：

```
var a = $('.red'); // 所有节点包含`class="red"`都将返回
// 例如:
// <div class="red">...</div>
// <p class="green red">...</p>
```

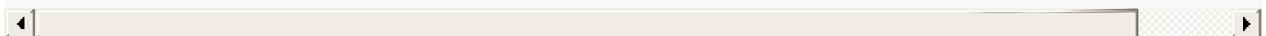
通常很多节点有多个class，我们可以查找同时包含 `red` 和 `green` 的节点：

```
var a = $('.red.green'); // 注意没有空格！
// 符合条件的节点：
// <div class="red green">...</div>
// <div class="blue green red">...</div>
```

按属性查找

一个DOM节点除了 `id` 和 `class` 外还可以有很多属性，很多时候按属性查找会非常方便，比如在一个表单中按属性来查找：

```
var email = $('[name=email]'); // 找出<??? name="email">
var passwordInput = $('[type=password]'); // 找出<??? type="passwor
var a = $('[items="A B"]'); // 找出<??? items="A B">
```



当属性的值包含空格等特殊字符时，需要用双引号括起来。

按属性查找还可以使用前缀查找或者后缀查找：

```
var icons = $('[name^=icon]'); // 找出所有name属性值以icon开头的DOM
// 例如: name="icon-1", name="icon-2"
var names = $('[name$=with]'); // 找出所有name属性值以with结尾的DOM
// 例如: name="startswith", name="endswith"
```

这个方法尤其适合通过class属性查找，且不受class包含多个名称的影响：

```
var icons = $('[class^="icon-"]'); // 找出所有class包含至少一个以`icon
// 例如: class="icon-clock", class="abc icon-home"
```

组合查找

组合查找就是把上述简单选择器组合起来使用。如果我们查找 `$('[name=email]')` ，很可能把表单外的 `<div name="email">` 也找出来，但我们只希望查找 `<input>` ，就可以这么写：

```
var emailInput = $('input[name=email]'); // 不会找出<div name="email
```

同样的，根据tag和class来组合查找也很常见：

```
var tr = $('tr.red'); // 找出<tr class="red ...">...</tr>
```

多项选择器

多项选择器就是把多个选择器用 `,` 组合起来一块选：

```
$('p,div'); // 把<p>和<div>都选出来
$('p.red,p.green'); // 把<p class="red">和<p class="green">都选出来
```

要注意的是，选出来的元素是按照它们在HTML中出现的顺序排列的，而且不会有重复元素。例如， `<p class="red green">` 不会被上面的 `$('p.red,p.green')` 选择两次。

练习

使用jQuery选择器分别选出指定元素：

- 仅选择JavaScript
- 仅选择Erlang
- 选择JavaScript和Erlang
- 选择所有编程语言
- 选择名字input
- 选择邮件和名字input

```
<!-- HTML结构 -->
<div id="test-jquery">
    <p id="para-1" class="color-red">JavaScript</p>
    <p id="para-2" class="color-green">Haskell</p>
    <p class="color-red color-green">Erlang</p>
    <p name="name" class="color-black">Python</p>
    <form class="test-form" target="_blank" action="#0" onsubmit="r
        <legend>注册新用户</legend>
        <fieldset>
            <p><label>名字: <input name="name"></label></p>
            <p><label>邮件: <input name="email"></label></p>
            <p><label>口令: <input name="password" type="password">
            <p><button type="submit">注册</button></p>
        </fieldset>
    </form>
</div>
```

运行查看结果：

```
'use strict';

var selected = null;

selected = ???;

// 高亮结果:
if (!(selected instanceof jQuery)) {
    return alert('不是有效的jQuery对象!');
}
$('#test-jquery').find('*').css('background-color', '');
selected.css('background-color', '#ffd351');
```

层级选择器

除了基本的选择器外，jQuery的层级选择器更加灵活，也更强大。

因为DOM的结构就是层级结构，所以我们经常要根据层级关系进行选择。

层级选择器（Descendant Selector）

如果两个DOM元素具有层级关系，就可以用 `$('ancestor descendant')` 来选择，层级之间用空格隔开。例如：

```
<!-- HTML结构 -->
<div class="testing">
    <ul class="lang">
        <li class="lang-javascript">JavaScript</li>
        <li class="lang-python">Python</li>
        <li class="lang-lua">Lua</li>
    </ul>
</div>
```

要选出JavaScript，可以用层级选择器：

```
$('ul.lang li.lang-javascript'); // [<li class="lang-javascript">Ja
$('div.testing li.lang-javascript'); // [<li class="lang-javascript
```

因为 <div> 和 <ul> 都是 <li> 的祖先节点，所以上面两种方式都可以选出相应的 <li> 节点。

要选择所有的 <li> 节点，用：

```
$('ul.lang li');
```

这种层级选择器相比单个的选择器好处在于，它缩小了选择范围，因为首先要定位父节点，才能选择相应的子节点，这样避免了页面其他不相关的元素。

例如：

```
$('form[name=upload] input');
```

就把选择范围限定在 `name` 属性为 `upload` 的表单里。如果页面有很多表单，其他表单的 `<input>` 不会被选择。

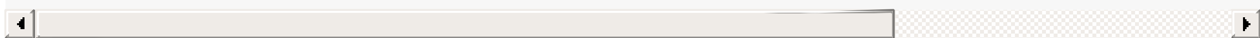
多层选择也是允许的：

```
$('form.test p input'); // 在form表单选择被<p>包含的<input>
```

子选择器（Child Selector）

子选择器 `$('parent>child')` 类似层级选择器，但是限定了层级关系必须是父子关系，就是 `<child>` 节点必须是 `<parent>` 节点的直属子节点。还是以上面的例子：

```
$('ul.lang>li.lang-javascript'); // 可以选出[<li class="lang-javascr
$('div.testing>li.lang-javascript'); // [], 无法选出, 因为<div>和<li>
```



过滤器（Filter）

过滤器一般不单独使用，它通常附加在选择器上，帮助我们更精确地定位元素。观察过滤器的效果：

```
$('ul.lang li'); // 选出JavaScript、Python和Lua 3个节点

$('ul.lang li:first-child'); // 仅选出JavaScript
$('ul.lang li:last-child'); // 仅选出Lua
$('ul.lang li:nth-child(2)'); // 选出第N个元素，N从1开始
$('ul.lang li:nth-child(even)'); // 选出序号为偶数的元素
$('ul.lang li:nth-child(odd)'); // 选出序号为奇数的元素
```

表单相关

针对表单元素，jQuery还有一组特殊的选择器：

- `:input`：可以选择`<input>`，`<textarea>`，`<select>`和`<button>`；
- `:file`：可以选择`<input type="file">`，和`input[type=file]`一样；
- `:checkbox`：可以选择复选框，和`input[type=checkbox]`一样；
- `:radio`：可以选择单选框，和`input[type=radio]`一样；
- `:focus`：可以选择当前输入焦点的元素，例如把光标放到一个`<input>`上，用`$('input:focus')`就可以选出；
- `:checked`：选择当前勾上的单选框和复选框，用这个选择器可以立刻获得用户选择的项目，如`$('input[type=radio]:checked')`；
- `:enabled`：可以选择可以正常输入的`<input>`、`<select>`等，也就是没有灰掉的输入；
- `:disabled`：和`:enabled`正好相反，选择那些不能输入的。

此外，jQuery还有很多有用的选择器，例如，选出可见的或隐藏的元素：

```
$('div:visible'); // 所有可见的div
$('div:hidden'); // 所有隐藏的div
```

练习

针对如下HTML结构：

```
<!-- HTML结构 -->

<div class="test-selector">
    <ul class="test-lang">
        <li class="lang-javascript">JavaScript</li>
        <li class="lang-python">Python</li>
        <li class="lang-lua">Lua</li>
    </ul>
    <ol class="test-lang">
        <li class="lang-swift">Swift</li>
        <li class="lang-java">Java</li>
        <li class="lang-c">C</li>
    </ol>
</div>
```

选出相应的内容并观察效果：

```
'use strict';
var selected = null;

// 分别选择所有语言，所有动态语言，所有静态语言，JavaScript，Lua，C等:
selected = ???

// 高亮结果:
if (!(selected instanceof jQuery)) {
    return alert('不是有效的jQuery对象!');
}
$('#test-jquery').find('*').css('background-color', '');
selected.css('background-color', '#ffd351');
```

查找和过滤

通常情况下选择器可以直接定位到我们想要的元素，但是，当我们拿到一个jQuery对象后，还可以以这个对象为基准，进行查找和过滤。

最常见的查找是在某个节点的所有子节点中查找，使用 `find()` 方法，它本身又接收一个任意的选择器。例如如下的HTML结构：

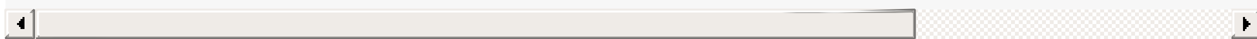
```
<!-- HTML结构 -->
<ul class="lang">
    <li class="js dy">JavaScript</li>
    <li class="dy">Python</li>
    <li id="swift">Swift</li>
    <li class="dy">Scheme</li>
    <li name="haskell">Haskell</li>
</ul>
```

用 `find()` 查找：

```
var ul = $('ul.lang'); // 获得<ul>
var dy = ul.find('.dy'); // 获得JavaScript, Python, Scheme
var swf = ul.find('#swift'); // 获得Swift
var hsk = ul.find('[name=haskell]'); // 获得Haskell
```

如果要从当前节点开始向上查找，使用 `parent()` 方法：

```
var swf = $('#swift'); // 获得Swift
var parent = swf.parent(); // 获得Swift的上层节点<ul>
var a = swf.parent('div.red'); // 从Swift的父节点开始向上查找，直到找到某
```



对于位于同一层级的节点，可以通过 `next()` 和 `prev()` 方法，例如：

当我们已经拿到 `Swift` 节点后：

```
var swift = $('#swift');

swift.next(); // Scheme
swift.next('[name=haskell]'); // Haskell，因为Haskell是后续第一个符合选

swift.prev(); // Python
swift.prev('.js'); // JavaScript，因为JavaScript是往前第一个符合选择器条
```

过滤

和函数式编程的map、filter类似，jQuery对象也有类似的方法。

`filter()` 方法可以过滤掉不符合选择器条件的节点：

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Sche
var a = langs.filter('.dy'); // 拿到JavaScript, Python, Scheme
```

或者传入一个函数，要特别注意函数内部的 `this` 被绑定为DOM对象，不是jQuery对象：

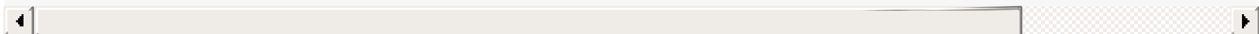
```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Sche
langs.filter(function () {
    return this.innerHTML.indexOf('S') === 0; // 返回S开头的节点
}); // 拿到Swift, Scheme
```

`map()` 方法把一个jQuery对象包含的若干DOM节点转化为其他对象：

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Sche
var arr = langs.map(function () {
    return this.innerHTML;
}).get(); // 用get()拿到包含string的Array：['JavaScript', 'Python', '
```

此外，一个jQuery对象如果包含了不止一个DOM节点，`first()`、`last()`和`slice()`方法可以返回一个新的jQuery对象，把不需要的DOM节点去掉：

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Sche
var js = langs.first(); // JavaScript，相当于$('ul.lang li:first-chi
var haskell = langs.last(); // Haskell, 相当于$('ul.lang li:last-chi
var sub = langs.slice(2, 4); // Swift, Scheme, 参数和数组的slice()方法
```



练习

对于下面的表单：

```
<form id="test-form" action="#0" onsubmit="return false;">
    <p><label>Name: <input name="name"></label></p>
    <p><label>Email: <input name="email"></label></p>
    <p><label>Password: <input name="password" type="password"></la
    <p>Gender: <label><input name="gender" type="radio" value="m" c
    <p><label>City: <select name="city">
        <option value="BJ" selected>Beijing</option>
        <option value="SH">Shanghai</option>
        <option value="CD">Chengdu</option>
        <option value="XM">Xiamen</option>
    </select></label></p>
    <p><button type="submit">Submit</button></p>
</form>
```



输入值后，用jQuery获取表单的JSON字符串，key和value分别对应每个输入的name和相应的value，例如：`{"name":"Michael","email":...}`

```
'use strict';
var json = null;

json = ???;

// 显示结果:
if (typeof(json) === 'string') {
    alert(json);
}
else {
    alert('json变量不是string!');
}
```

操作DOM

jQuery的选择器很强大，用起来又简单又灵活，但是搞了这么久，我拿到了jQuery对象，到底要干什么？

答案当然是操作对应的DOM节点啦！

回顾一下修改DOM的CSS、文本、设置HTML有多么麻烦，而且有的浏览器只有innerHTML，有的浏览器支持innerText，有了jQuery对象，不需要考虑浏览器差异了，全部统一操作！

修改Text和HTML

jQuery对象的 `text()` 和 `html()` 方法分别获取节点的文本和原始HTML文本，例如，如下的HTML结构：

```
<!-- HTML结构 -->
<ul id="test-ul">
    <li class="js">JavaScript</li>
    <li name="book">Java &amp; JavaScript</li>
</ul>
```

分别获取文本和HTML：

```
$('#test-ul li[name=book]').text(); // 'Java & JavaScript'
$('#test-ul li[name=book]').html(); // 'Java &amp; JavaScript'
```

如何设置文本或HTML？jQuery的API设计非常巧妙：无参数调用 `text()` 是获取文本，传入参数就变成设置文本，HTML也是类似操作，自己动手试试：

```
'use strict';
var j1 = $('#test-ul li.js');
var j2 = $('#test-ul li[name=book]');

j1.html('<span style="color: red">JavaScript</span>');
j2.text('JavaScript & ECMAScript');
```

一个jQuery对象可以包含0个或任意个DOM对象，它的方法实际上会作用在对应的每个DOM节点上。在上面的例子中试试：

```
$('#test-ul li').text('JS'); // 是不是两个节点都变成了JS？
```

所以jQuery对象的另一个好处是我们可以执行一个操作，作用在对应的一组DOM节点上。即使选择器没有返回任何DOM节点，调用jQuery对象的方法仍然不会报错：

```
// 如果不存在id为not-exist的节点：
$('#not-exist').text('Hello'); // 代码不报错，没有节点被设置为'Hello'
```

这意味着jQuery帮你免去了许多 `if` 语句。

修改CSS

jQuery对象有“批量操作”的特点，这用于修改CSS实在是太方便了。考虑下面的HTML结构：

```
<!-- HTML结构 -->
<ul id="test-css">
    <li class="lang dy"><span>JavaScript</span></li>
    <li class="lang"><span>Java</span></li>
    <li class="lang dy"><span>Python</span></li>
    <li class="lang"><span>Swift</span></li>
    <li class="lang dy"><span>Scheme</span></li>
</ul>
```

要高亮显示动态语言，调用jQuery对象的`css('name', 'value')`方法，我们用一行语句实现：

```
'use strict';

$('#test-css li.dy>span').css('background-color', '#ffd351').css('c
```

注意，jQuery对象的所有方法都返回一个jQuery对象（可能是新的也可能是自身），这样我们可以进行链式调用，非常方便。

jQuery对象的`css()`方法可以这么用：

```
var div = $('#test-div');
div.css('color'); // '#000033', 获取CSS属性
div.css('color', '#336699'); // 设置CSS属性
div.css('color', ''); // 清除CSS属性
```

为了和JavaScript保持一致，CSS属性可以用`'background-color'`和`'backgroundColor'`两种格式。

`css()`方法将作用于DOM节点的`style`属性，具有最高优先级。如果要修改`class`属性，可以用jQuery提供的下列方法：

```
var div = $('#test-div');
div.hasClass('highlight'); // false， class是否包含highlight
div.addClass('highlight'); // 添加highlight这个class
div.removeClass('highlight'); // 删除highlight这个class
```

练习：分别用`css()`方法和`addClass()`方法高亮显示JavaScript：

```
<!-- HTML结构 -->
<style>
.highlight {
    color: #dd1144;
    background-color: #ffd351;
}
</style>

<div id="test-highlight-css">
    <ul>
        <li class="py"><span>Python</span></li>
        <li class="js"><span>JavaScript</span></li>
        <li class="sw"><span>Swift</span></li>
        <li class="hk"><span>Haskell</span></li>
    </ul>
</div>
```

```
'use strict';

var div = $('#test-highlight-css');
// TODO:
```

显示和隐藏DOM

要隐藏一个DOM，我们可以设置CSS的 `display` 属性为 `none` ，利用 `css()` 方法就可以实现。不过，要显示这个DOM就需要恢复原有的 `display` 属性，这就得先记下来原有的 `display` 属性到底是 `block` 还是 `inline` 还是别的值。

考虑到显示和隐藏DOM元素使用非常普遍，jQuery直接提供 `show()` 和 `hide()` 方法，我们不用关心它是如何修改 `display` 属性的，总之它能正常工作：

```
var a = $('a[target=_blank]');
a.hide(); // 隐藏
a.show(); // 显示
```

注意，隐藏DOM节点并未改变DOM树的结构，它只影响DOM节点的显示。这和删除DOM节点是不同的。

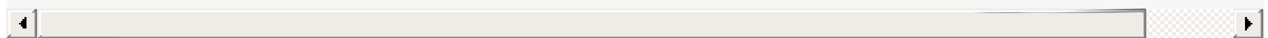
获取DOM信息

利用jQuery对象的若干方法，我们直接可以获取DOM的高宽等信息，而无需针对不同浏览器编写特定代码：

```
// 浏览器可视窗口大小:
$(window).width(); // 800
$(window).height(); // 600

// HTML文档大小:
$(document).width(); // 800
$(document).height(); // 3500

// 某个div的大小:
var div = $('#test-div');
div.width(); // 600
div.height(); // 300
div.width(400); // 设置CSS属性 width: 400px，是否生效要看CSS是否有效
div.height('200px'); // 设置CSS属性 height: 200px，是否生效要看CSS是否有
```

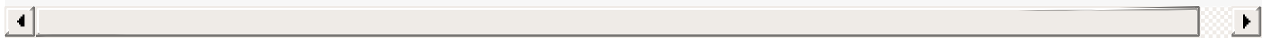


`attr()` 和 `removeAttr()` 方法用于操作DOM节点的属性：

```
// <div id="test-div" name="Test" start="1">...</div>
var div = $('#test-div');
div.attr('data'); // undefined, 属性不存在
div.attr('name'); // 'Test'
div.attr('name', 'Hello'); // div的name属性变为'Hello'
div.removeAttr('name'); // 删除name属性
div.attr('name'); // undefined
```

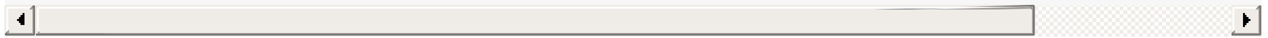
`prop()` 方法和 `attr()` 类似，但是HTML5规定有一种属性在DOM节点中可以没有值，只有出现与不出现两种，例如：

```
<input id="test-radio" type="radio" name="test" checked value="1">
```



等价于：

```
<input id="test-radio" type="radio" name="test" checked="checked" v
```



`attr()` 和 `prop()` 对于属性 `checked` 处理有所不同：

```
var radio = $('#test-radio');
radio.attr('checked'); // 'checked'
radio.prop('checked'); // true
```

`prop()` 返回值更合理一些。不过，用 `is()` 方法判断更好：

```
var radio = $('#test-radio');
radio.is(':checked'); // true
```

类似的属性还有 `selected` ，处理时最好用 `is(':selected')` 。

操作表单

对于表单元素，jQuery对象统一提供 `val()` 方法获取和设置对应的 `value` 属性：

```
/*
    <input id="test-input" name="email" value="">
    <select id="test-select" name="city">
        <option value="BJ" selected>Beijing</option>
        <option value="SH">Shanghai</option>
        <option value="SZ">Shenzhen</option>
    </select>
    <textarea id="test-textarea">Hello</textarea>
*/
var
    input = $('#test-input'),
    select = $('#test-select'),
    textarea = $('#test-textarea');

input.val(); // 'test'
input.val('abc@example.com'); // 文本框的内容已变为abc@example.com

select.val(); // 'BJ'
select.val('SH'); // 选择框已变为Shanghai

textarea.val(); // 'Hello'
textarea.val('Hi'); // 文本区域已更新为'Hi'
```

可见，一个 `val()` 就统一了各种输入框的取值和赋值的问题。

修改DOM结构

直接使用浏览器提供的API对DOM结构进行修改，不但代码复杂，而且要针对浏览器写不同的代码。

有了jQuery，我们就专注于操作jQuery对象本身，底层的DOM操作由jQuery完成就可以了，这样一来，修改DOM也大大简化了。

添加DOM

要添加新的DOM节点，除了通过jQuery的 `html()` 这种暴力方法外，还可以用 `append()` 方法，例如：

```
<div id="test-div">
    <ul>
        <li><span>JavaScript</span></li>
        <li><span>Python</span></li>
        <li><span>Swift</span></li>
    </ul>
</div>
```

如何向列表新增一个语言？首先要拿到 `<ul>` 节点：

```
var ul = $('#test-div>ul');
```

然后，调用 `append()` 传入HTML片段：

```
ul.append('<li><span>Haskell</span></li>');
```

除了接受字符串， `append()` 还可以传入原始的DOM对象，jQuery对象和函数对象：

```
// 创建DOM对象:
var ps = document.createElement('li');
ps.innerHTML = '<span>Pascal</span>';
// 添加DOM对象:
ul.append(ps);

// 添加jQuery对象:
ul.append($('#scheme'));

// 添加函数对象:
ul.append(function (index, html) {
    return '<li><span>Language - ' + index + '</span></li>';
});
```

传入函数时，要求返回一个字符串、DOM对象或者jQuery对象。因为jQuery的 `append()` 可能作用于一组DOM节点，只有传入函数才能针对每个DOM生成不同的子节点。

`append()` 把DOM添加到最后，`prepend()` 则把DOM添加到最前。

另外注意，如果要添加的DOM节点已经存在于HTML文档中，它会首先从文档移除，然后再添加，也就是说，用 `append()` ，你可以移动一个DOM节点。

如果要把新节点插入到指定位置，例如，JavaScript和Python之间，那么，可以先定位到JavaScript，然后用 `after()` 方法：

```
var js = $('#test-div>ul>li:first-child');
js.after('<li><span>Lua</span></li>');
```

也就是说，同级节点可以用 `after()` 或者 `before()` 方法。

删除节点

要删除DOM节点，拿到jQuery对象后直接调用 `remove()` 方法就可以了。如果jQuery对象包含若干DOM节点，实际上可以一次删除多个DOM节点：

```
var li = $('#test-div>ul>li');
li.remove(); // 所有<li>全被删除
```

练习

除了列出的3种语言外，请再添加Pascal、Lua和Ruby，然后按字母顺序排序节点：

```
<!-- HTML结构 -->
<div id="test-div">
    <ul>
        <li><span>JavaScript</span></li>
        <li><span>Python</span></li>
        <li><span>Swift</span></li>
    </ul>
</div>
```

```
'use strict';



// 测试:
;(function () {
    var s = $('#test-div>ul>li').map(function () {
        return $(this).text();
    }).get().join(',');
    if (s === 'JavaScript,Lua,Pascal,Python,Ruby,Swift') {
        alert('测试通过!');
    } else {
        alert('测试失败: ' + s);
    }
})();
```

事件

因为JavaScript在浏览器中以单线程模式运行，页面加载后，一旦页面上所有的JavaScript代码被执行完后，就只能依赖触发事件来执行JavaScript代码。

浏览器在接收到用户的鼠标或键盘输入后，会自动在对应的DOM节点上触发相应的事件。如果该节点已经绑定了对应的JavaScript处理函数，该函数就会自动调用。

由于不同的浏览器绑定事件的代码都不太一样，所以用jQuery来写代码，就屏蔽了不同浏览器的差异，我们总是编写相同的代码。

举个例子，假设要在用户点击了超链接时弹出提示框，我们用jQuery这样绑定一个 `click` 事件：

```
/* HTML:
 *
 * <a id="test-link" href="#0">点我试试</a>
 *
 */

// 获取超链接的jQuery对象:
var a = $('#test-link');
a.on('click', function () {
    alert('Hello!');
});
```

实测：点我试试

`on` 方法用来绑定一个事件，我们需要传入事件名称和对应的处理函数。

另一种更简化的写法是直接调用 `click()` 方法：

```
a.click(function () {
    alert('Hello!');
});
```

两者完全等价。我们通常用后面的写法。

jQuery能够绑定的事件主要包括：

鼠标事件

click: 鼠标单击时触发； dblclick：鼠标双击时触发； mouseenter：鼠标进入时触发； mouseleave：鼠标移出时触发； mousemove：鼠标在DOM内部移动时触发； hover：鼠标进入和退出时触发两个函数，相当于mouseenter加上mouseleave。

键盘事件

键盘事件仅作用在当前焦点的DOM上，通常是 `<input>` 和 `<textarea>` 。

keydown：键盘按下时触发； keyup：键盘松开时触发； keypress：按一次键后触发。

其他事件

focus：当DOM获得焦点时触发； blur：当DOM失去焦点时触发； change：当 `<input>` 、 `<select>` 或 `<textarea>` 的内容改变时触发； submit：当 `<form>` 提交时触发； ready：当页面被载入并且DOM树完成初始化后触发。

其中， `ready` 仅作用于 `document` 对象。由于 `ready` 事件在DOM完成初始化后触发，且只触发一次，所以非常适合用来写其他的初始化代码。假设我们想给一个 `<form>` 表单绑定 `submit` 事件，下面的代码没有预期的效果：

```
<html>
<head>
    <script>
        // 代码有误:
        $('#testForm).on('submit', function () {
            alert('submit!');
        });
    </script>
</head>
<body>
    <form id="testForm">
        ...
    </form>
</body>
```

因为JavaScript在此执行的时候，`<form>` 尚未载入浏览器，所以 `$('#testForm)` 返回 `[]`，并没有绑定事件到任何DOM上。

所以我们自己的初始化代码必须放到 `document` 对象的 `ready` 事件中，保证DOM已完成初始化：

```
<html>
<head>
    <script>
        $(document).on('ready', function () {
            $('#testForm).on('submit', function () {
                alert('submit!');
            });
        });
    </script>
</head>
<body>
    <form id="testForm">
        ...
    </form>
</body>
```

这样写就没有问题了。因为相关代码会在DOM树初始化后再执行。

由于 `ready` 事件使用非常普遍，所以可以这样简化：

```
$(document).ready(function () {
    // on('submit', function)也可以简化:
    $('#testForm).submit(function () {
        alert('submit!');
    });
});
```

甚至还可以再简化为：

```
$(function () {
    // init...
});
```

上面的这种写法最为常见。如果你遇到 `$(function () {...})` 的形式，牢记这是 `document` 对象的 `ready` 事件处理函数。

完全可以反复绑定事件处理函数，它们会依次执行：

```
$(function () {
    console.log('init A...');
});
$(function () {
    console.log('init B...');
});
$(function () {
    console.log('init C...');
});
```

事件参数

有些事件，如 `mousemove` 和 `keypress` ，我们需要获取鼠标位置和按键的值，否则监听这些事件就没什么意义了。所有事件都会传入 `Event` 对象作为参数，可以从 `Event` 对象上获取到更多的信息：

```
$(function () {
    $('#testMouseMoveDiv').mousemove(function (e) {
        $('#testMouseMoveSpan').text('pageX = ' + e.pageX + ', page
    });
});
```

取消绑定

一个已被绑定的事件可以解除绑定，通过 `off('click', function)` 实现：

```
function hello() {
    alert('hello!');
}

a.click(hello); // 绑定事件

// 10秒钟后解除绑定:
setTimeout(function () {
    a.off('click', hello);
}, 10000);
```

需要特别注意的是，下面这种写法是无效的：

```
// 绑定事件:
a.click(function () {
    alert('hello!');
});

// 解除绑定:
a.off('click', function () {
    alert('hello!');
});
```

这是因为两个匿名函数虽然长得一模一样，但是它们是两个不同的函数对象， `off('click', function () {...})` 无法移除已绑定的第一个匿名函数。

为了实现移除效果，可以使用 `off('click')` 一次性移除已绑定的 `click` 事件的所有处理函数。

同理，无参数调用 `off()` 一次性移除已绑定的所有类型的事件处理函数。

事件触发条件

一个需要注意的问题是，事件的触发总是由用户操作引发的。例如，我们监控文本框的内容改动：

```
var input = $('#test-input');
input.change(function () {
    console.log('changed...');
});
```

当用户在文本框中输入时，就会触发 `change` 事件。但是，如果用JavaScript代码去改动文本框的值，将不会触发 `change` 事件：

```
var input = $('#test-input');
input.val('change it!'); // 无法触发change事件
```

有些时候，我们希望用代码触发 `change` 事件，可以直接调用无参数的 `change()` 方法来触发该事件：

```
var input = $('#test-input');
input.val('change it!');
input.change(); // 触发change事件
```

`input.change()` 相当于 `input.trigger('change')`，它是 `trigger()` 方法的简写。

为什么我们希望手动触发一个事件呢？如果不这么做，很多时候，我们就得写两份一模一样的代码。

浏览器安全限制

在浏览器中，有些JavaScript代码只有在用户触发下才能执行，例如，`window.open()`函数：

```
// 无法弹出新窗口，将被浏览器屏蔽:
$(function () {
    window.open('/');
});
```

这些“敏感代码”只能由用户操作来触发：

```
var button1 = $('#testPopupButton1');
var button2 = $('#testPopupButton2');

function popupTestWindow() {
    window.open('/');
}

button1.click(function () {
    popupTestWindow();
});

button2.click(function () {
    // 不立刻执行popupTestWindow()，100毫秒后执行:
    setTimeout(popupTestWindow, 100);
});
```

当用户点击`button1`时，`click`事件被触发，由于`popupTestWindow()`在`click`事件处理函数内执行，这是浏览器允许的，而`button2`的`click`事件并未立刻执行`popupTestWindow()`，延迟执行的`popupTestWindow()`将被浏览器拦截。

练习

对如下的Form表单：

```
<!-- HTML结构 -->
<form id="test-form" action="test">
    <legend>请选择想要学习的编程语言：</legend>
    <fieldset>
        <p><label class="selectAll"><input type="checkbox"> <span c
        <p><label><input type="checkbox" name="lang" value="javascr
        <p><label><input type="checkbox" name="lang" value="python"
        <p><label><input type="checkbox" name="lang" value="ruby"> 
        <p><label><input type="checkbox" name="lang" value="haskell
        <p><label><input type="checkbox" name="lang" value="scheme"
        <p><button type="submit">Submit</button></p>
    </fieldset>
</form>
```

绑定合适的事件处理函数，实现以下逻辑：

当用户勾上“全选”时，自动选中所有语言，并把“全选”变成“全不选”；

当用户去掉“全不选”时，自动不选中所有语言；

当用户点击“反选”时，自动把所有语言状态反转（选中的变为未选，未选的变为选中）；

当用户把所有语言都手动勾上时，“全选”被自动勾上，并变为“全不选”；

当用户手动去掉选中至少一种语言时，“全不选”自动被去掉选中，并变为“全选”。

```
'use strict';

var
    form = $('#test-form'),
    langs = form.find('[name=lang]'),
    selectAll = form.find('label.selectAll :checkbox'),
    selectAllLabel = form.find('label.selectAll span.selectAll'),
    deselectAllLabel = form.find('label.selectAll span.deselectAll
    invertSelect = form.find('a.invertSelect');

// 重置初始化状态:
form.find('*').show().off();
form.find(':checkbox').prop('checked', false).off();
deselectAllLabel.hide();
// 拦截form提交事件:
form.off().submit(function (e) {
    e.preventDefault();
    alert(form.serialize());
});

// TODO:绑定事件

// 测试:
alert('请测试功能是否正常。');
```

动画

用JavaScript实现动画，原理非常简单：我们只需要以固定的时间间隔（例如，0.1秒），每次把DOM元素的CSS样式修改一点（例如，高宽各增加10%），看起来就像动画了。

但是要用JavaScript手动实现动画效果，需要编写非常复杂的代码。如果想要把动画效果用函数封装起来便于复用，那考虑的事情就更多了。

使用jQuery实现动画，代码已经简单得不能再简化了：只需要一行代码！

让我们先来看看jQuery内置的几种动画样式：

show / hide

直接以无参数形式调用 `show()` 和 `hide()` ，会显示和隐藏DOM元素。但是，只要传递一个时间参数进去，就变成了动画：

```
var div = $('#test-show-hide');
div.hide(3000); // 在3秒钟内逐渐消失
```

时间以毫秒为单位，但也可以是 `'slow'` ， `'fast'` 这些字符串：

```
var div = $('#test-show-hide');
div.show('slow'); // 在0.6秒钟内逐渐显示
```

`toggle()` 方法则根据当前状态决定是 `show()` 还是 `hide()` 。

slideUp / slideDown

你可能已经看出来了， `show()` 和 `hide()` 是从左上角逐渐展开或收缩的，而 `slideUp()` 和 `slideDown()` 则是在垂直方向逐渐展开或收缩的。

`slideUp()` 把一个可见的DOM元素收起来，效果跟拉上窗帘似的，`slideDown()` 相反，而 `slideToggle()` 则根据元素是否可见来决定下一步动作：

```
var div = $('#test-slide');
div.slideUp(3000); // 在3秒钟内逐渐向上消失
```

fadeIn / fadeOut

`fadeIn()` 和 `fadeOut()` 的动画效果是淡入淡出，也就是通过不断设置DOM元素的 `opacity` 属性来实现，而 `fadeToggle()` 则根据元素是否可见来决定下一步动作：

```
var div = $('#test-fade');
div.fadeOut('slow'); // 在0.6秒内淡出
```

fadeOut('slow') fadeIn('slow') fadeToggle('slow')

自定义动画

如果上述动画效果还不能满足你的要求，那就祭出最后大招：`animate()`，它可以实现任意动画效果，我们需要传入的参数就是DOM元素最终的CSS状态和时间，jQuery在时间段内不断调整CSS直到达到我们设定的值：

```
var div = $('#test-animate');
div.animate({
    opacity: 0.25,
    width: '256px',
    height: '256px'
}, 3000); // 在3秒钟内CSS过渡到设定值
```

`animate()` 还可以再传入一个函数，当动画结束时，该函数将被调用：

```
var div = $('#test-animate');
div.animate({
    opacity: 0.25,
    width: '256px',
    height: '256px'
}, 3000, function () {
    console.log('动画已结束');
    // 恢复至初始状态:
    $(this).css('opacity', '1.0').css('width', '128px').css('height
});
```

实际上这个回调函数参数对于基本动画也是适用的。

有了 `animate()`，你就可以实现各种自定义动画效果了：

串行动画

jQuery的动画效果还可以串行执行，通过 `delay()` 方法还可以实现暂停，这样，我们可以实现更复杂的动画效果，而代码却相当简单：

```
var div = $('#test-animates');
// 动画效果：slideDown - 暂停 - 放大 - 暂停 - 缩小
div.slideDown(2000)
   .delay(1000)
   .animate({
       width: '256px',
       height: '256px'
   }, 2000)
   .delay(1000)
   .animate({
       width: '128px',
       height: '128px'
   }, 2000);
}
</script>
```

因为动画需要执行一段时间，所以jQuery必须不断返回新的Promise对象才能后续执行操作。简单地把动画封装在函数中是不够的。

为什么有的动画没有效果

你可能会遇到，有的动画如 `slideUp()` 根本没有效果。这是因为jQuery动画的原理是逐渐改变CSS的值，如 `height` 从 `100px` 逐渐变为 `0` 。但是很多不是block性质的DOM元素，对它们设置 `height` 根本就不起作用，所以动画也就没有效果。

此外，jQuery也没有实现对 `background-color` 的动画效果，用 `animate()` 设置 `background-color` 也没有效果。这种情况下可以使用CSS3的 `transition` 实现动画效果。

练习

在执行删除操作时，给用户显示一个动画比直接调用 `remove()` 要更好。请在表格删除一行的时候添加一个淡出的动画效果：

```
'use strict';

function deleteFirstTR() {
    var tr = $('#test-table>tbody>tr:visible').first();



}

deleteFirstTR();
```

AJAX

用JavaScript写AJAX前面已经介绍过了，主要问题就是不同浏览器需要写不同代码，并且状态和错误处理写起来很麻烦。

用jQuery的相关对象来处理AJAX，不但不需要考虑浏览器问题，代码也能大大简化。

ajax

jQuery在全局对象 `jQuery` （也就是 `$` ）绑定了 `ajax()` 函数，可以处理AJAX请求。 `ajax(url, settings)` 函数需要接收一个URL和一个可选的 `settings` 对象，常用的选项如下：

- async：是否异步执行AJAX请求，默认为 `true` ，千万不要指定为 `false` ；
- method：发送的Method，缺省为 `'GET'` ，可指定为 `'POST'` 、 `'PUT'` 等；
- contentType：发送POST请求的格式，默认值为 `'application/x-www-form-urlencoded; charset=UTF-8'` ，也可以指定为 `text/plain` 、 `application/json` ；
- data：发送的数据，可以是字符串、数组或object。如果是GET请求，data将被转换成query附加到URL上，如果是POST请求，根据contentType把data序列化成合适的格式；
- headers：发送的额外的HTTP头，必须是一个object；
- dataType：接收的数据格式，可以指定为 `'html'` 、 `'xml'` 、 `'json'` 、 `'text'` 等，缺省情况下根据响应的 `Content-Type` 猜测。

下面的例子发送一个GET请求，并返回一个JSON格式的数据：

```
var jqxhr = $.ajax('/api/categories', {
    dataType: 'json'
});
// 请求已经发送了
```

不过，如何用回调函数处理返回的数据和出错时的响应呢？

还记得Promise对象吗？jQuery的jqXHR对象类似一个Promise对象，我们可以用链式写法来处理各种回调：

```
'use strict';

function ajaxLog(s) {
    var txt = $('#test-response-text');
    txt.val(txt.val() + '\n' + s);
}

$('#test-response-text').val('');

var jqxhr = $.ajax('/api/categories', {
    dataType: 'json'
}).done(function (data) {
    ajaxLog('成功, 收到的数据: ' + JSON.stringify(data));
}).fail(function (xhr, status) {
    ajaxLog('失败: ' + xhr.status + ', 原因: ' + status);
}).always(function () {
    ajaxLog('请求完成: 无论成功或失败都会调用');
});
```

get

对常用的AJAX操作，jQuery提供了一些辅助方法。由于GET请求最常见，所以jQuery提供了 `get()` 方法，可以这么写：

```
var jqxhr = $.get('/path/to/resource', {
    name: 'Bob Lee',
    check: 1
});
```

第二个参数如果是object，jQuery自动把它变成query string然后加到URL后面，实际的URL是：

```
/path/to/resource?name=Bob%20Lee&check=1
```

这样我们就不用关心如何用URL编码并构造一个query string了。

post

`post()` 和 `get()` 类似，但是传入的第二个参数默认被序列化为 `application/x-www-form-urlencoded` ：

```
var jqxhr = $.post('/path/to/resource', {
    name: 'Bob Lee',
    check: 1
});
```

实际构造的数据 `name=Bob%20Lee&check=1` 作为POST的body被发送。

getJSON

由于JSON用得越来越普遍，所以jQuery也提供了 `getJSON()` 方法来快速通过GET获取一个JSON对象：

```
var jqxhr = $.getJSON('/path/to/resource', {
    name: 'Bob Lee',
    check: 1
}).done(function (data) {
    // data已经被解析为JSON对象了
});
```

安全限制

jQuery的AJAX完全封装的是JavaScript的AJAX操作，所以它的安全限制和前面讲的用JavaScript写AJAX完全一样。

如果需要使用JSONP，可以在 `ajax()` 中设置 `jsonp: 'callback'` ，让jQuery实现JSONP跨域加载数据。

关于跨域的设置请参考浏览器 - AJAX一节中CORS的设置。

扩展

当我们使用jQuery对象的方法时，由于jQuery对象可以操作一组DOM，而且支持链式操作，所以用起来非常方便。

但是jQuery内置的方法永远不可能满足所有的需求。比如，我们想要高亮显示某些DOM元素，用jQuery可以这么实现：

```
$('span.hl').css('backgroundColor', '#fffceb').css('color', '#d8503

$('p a.hl').css('backgroundColor', '#fffceb').css('color', '#d8503(
```

总是写重复代码可不好，万一以后还要修改字体就更麻烦了，能不能统一起来，写个 `highlight()` 方法？

```
$('span.hl').highlight();

$('p a.hl').highlight();
```

答案是肯定的。我们可以扩展jQuery来实现自定义方法。将来如果要修改高亮的逻辑，只需修改一处扩展代码。这种方式也称为编写jQuery插件。

编写jQuery插件

给jQuery对象绑定一个新方法是通过扩展 `$.fn` 对象实现的。让我们来编写第一个扩展—— `highlight1()` ：

```
$.fn.highlight1 = function () {
    // this已绑定为当前jQuery对象:
    this.css('backgroundColor', '#fffceb').css('color', '#d85030');
    return this;
}
```

注意到函数内部的 `this` 在调用时被绑定为jQuery对象，所以函数内部代码可以正常调用所有jQuery对象的方法。

对于如下的HTML结构：

```
<!-- HTML结构 -->
<div id="test-highlight1">
    <p>什么是<span>jQuery</span></p>
    <p><span>jQuery</span>是目前最流行的<span>JavaScript</span>库。</p
</div>
```

来测试一下 `highlight1()` 的效果：

```
'use strict';

$('#test-highlight1 span').highlight1();
```

细心的童鞋可能发现了，为什么最后要 `return this;` ？因为jQuery对象支持链式操作，我们自己写的扩展方法也要能继续链式下去：

```
$('span.hl').highlight1().slideDown();
```

不然，用户调用的时候，就不得不把上面的代码拆成两行。

但是这个版本并不完美。有的用户希望高亮的颜色能自己来指定，怎么办？

我们可以给方法加个参数，让用户自己把参数用对象传进去。于是我们有了第二个版本的 `highlight2()` ：

```
$.fn.highlight2 = function (options) {
    // 要考虑到各种情况:
    // options为undefined
    // options只有部分key
    var bgcolor = options && options.backgroundColor || '#fffceb';
    var color = options && options.color || '#d85030';
    this.css('backgroundColor', bgcolor).css('color', color);
    return this;
}
```

对于如下HTML结构：

```
<!-- HTML结构 -->
<div id="test-highlight2">
    <p>什么是<span>jQuery</span> <span>Plugin</span></p>
    <p>编写<span>jQuery</span> <span>Plugin</span>可以用来扩展<span>j
</div>
```

来实测一下带参数的 `highlight2()` ：

```
'use strict';

$('#test-highlight2 span').highlight2({
    backgroundColor: '#00a8e6',
    color: '#ffffff'
});
```

对于默认值的处理，我们用了一个简单的 `&&` 和 `||` 短路操作符，总能得到一个有效的值。

另一种方法是使用jQuery提供的辅助方法 `$.extend(target, obj1, obj2, ...)` ，它把多个object对象的属性合并到第一个target对象中，遇到同名属性，总是使用靠后的对象的值，也就是越往后优先级越高：

```
// 把默认值和用户传入的options合并到对象{}中并返回:
var opts = $.extend({}, {
    backgroundColor: '#00a8e6',
    color: '#ffffff'
}, options);
```

紧接着用户对 `highlight2()` 提出了意见：每次调用都需要传入自定义的设置，能不能让我自己设定一个缺省值，以后的调用统一使用无参数的 `highlight2()` ？

也就是说，我们设定的默认值应该能允许用户修改。

那默认值放哪比较合适？放全局变量肯定不合适，最佳地点是 `$.fn.highlight2` 这个函数对象本身。

于是最终版的 `highlight()` 终于诞生了：

```
$.fn.highlight = function (options) {
    // 合并默认值和用户设定值:
    var opts = $.extend({}, $.fn.highlight.defaults, options);
    this.css('backgroundColor', opts.backgroundColor).css('color',
    return this;
}

// 设定默认值:
$.fn.highlight.defaults = {
    color: '#d85030',
    backgroundColor: '#fff8de'
}
```

这次用户终于满意了。用户使用时，只需一次性设定默认值：

```
$.fn.highlight.defaults.color = '#fff';
$.fn.highlight.defaults.backgroundColor = '#000';
```

然后就可以非常简单地调用 `highlight()` 了。

对如下的HTML结构：

```
<!-- HTML结构 -->
<div id="test-highlight">
    <p>如何编写<span>jQuery</span> <span>Plugin</span></p>
    <p>编写<span>jQuery</span> <span>Plugin</span>，要设置<span>默认值
</div>
```

实测一下修改默认值的效果：

```
'use strict';

$.fn.highlight.defaults.color = '#659f13';
$.fn.highlight.defaults.backgroundColor = '#f2fae3';

$('#test-highlight p:first-child span').highlight();

$('#test-highlight p:last-child span').highlight({
    color: '#dd1144'
});
```

最终，我们得出编写一个jQuery插件的原则：

1. 给 `$.fn` 绑定函数，实现插件的代码逻辑；
2. 插件函数最后要 `return this;` 以支持链式调用；
3. 插件函数要有默认值，绑定在 `$.fn.<pluginName>.defaults` 上；
4. 用户在调用时可传入设定值以便覆盖默认值。

针对特定元素的扩展

我们知道jQuery对象的有些方法只能作用在特定DOM元素上，比如 `submit()` 方法只能针对 `form` 。如果我们编写的扩展只能针对某些类型的DOM元素，应该怎么写？

还记得jQuery的选择器支持 `filter()` 方法来过滤吗？我们可以借助这个方法来实现针对特定元素的扩展。

举个例子，现在我们要给所有指向外链的超链接加上跳转提示，怎么做？

先写出用户调用的代码：

```
$('#main a').external();
```

然后按照上面的方法编写一个 `external` 扩展：

```
$.fn.external = function () {
    // return返回的each()返回结果，支持链式调用:
    return this.filter('a').each(function () {
        // 注意: each()内部的回调函数的this绑定为DOM本身!
        var a = $(this);
        var url = a.attr('href');
        if (url && (url.indexOf('http://')===0 || url.indexOf('http
            a.attr('href', '#0')
             .removeAttr('target')
             .append(' <i class="uk-icon-external-link"></i>')
             .click(function () {
                if(confirm('你确定要前往' + url + '？')) {
                    window.open(url);
                }
            });
        }
    });
}
```

对如下的HTML结构：

```
<!-- HTML结构 -->
<div id="test-external">
    <p>如何学习<a href="http://jquery.com">jQuery</a>？</p>
    <p>首先，你要学习<a href="/wiki/001434446689867b27157e896e74d51a8
</div>
```

实测外链效果：

```
'use strict';

$('#test-external a').external();
```

小结

扩展jQuery对象的功能十分简单，但是我们要遵循jQuery的原则，编写的扩展方法能支持链式调用、具备默认值和过滤特定元素，使得扩展方法看上去和jQuery本身的方法没有什么区别。

underscore

前面我们已经讲过了，JavaScript是函数式编程语言，支持高阶函数和闭包。函数式编程非常强大，可以写出非常简洁的代码。例如 `Array` 的 `map()` 和 `filter()` 方法：

```
'use strict';
var a1 = [1, 4, 9, 16];
var a2 = a1.map(Math.sqrt); // [1, 2, 3, 4]
var a3 = a2.filter((x) => { return x % 2 === 0; }); // [2, 4]
```

现在问题来了， `Array` 有 `map()` 和 `filter()` 方法，可是Object没有这些方法。此外，低版本的浏览器例如IE6～8也没有这些方法，怎么办？

方法一，自己把这些方法添加到 `Array.prototype` 中，然后给 `Object.prototype` 也加上 `mapObject()` 等类似的方法。

方法二，直接找一个成熟可靠的第三方开源库，使用统一的函数来实现 `map()` 、 `filter()` 这些操作。

我们采用方法二，选择的第三方库就是underscore。

正如jQuery统一了不同浏览器之间的DOM操作的差异，让我们可以简单地对DOM进行操作，underscore则提供了一套完善的函数式编程的接口，让我们更方便地在JavaScript中实现函数式编程。

jQuery在加载时，会把自身绑定到唯一的全局变量 `$` 上，underscore与其类似，会把自身绑定到唯一的全局变量 `_` 上，这也是为啥它的名字叫underscore的原因。

用underscore实现 `map()` 操作如下：

```
'use strict';
_.map([1, 2, 3], (x) => x * x); // [1, 4, 9]
```

咋一看比直接用 `Array.map()` 要麻烦一点，可是underscore的 `map()` 还可以作用于Object：

```
'use strict';
_.map({ a: 1, b: 2, c: 3 }, (v, k) => k + '=' + v); // ['a=1', 'b=2
```

后面我们会详细介绍underscore提供了一系列函数式接口。

Collections

underscore为集合类对象提供了一致的接口。集合类是指Array和Object，暂不支持Map和Set。

map/filter

和 `Array` 的 `map()` 与 `filter()` 类似，但是underscore的 `map()` 和 `filter()` 可以作用于Object。当作用于Object时，传入的函数为 `function (value, key)` ，第一个参数接收value，第二个参数接收key：

```
'use strict';

var obj = {
    name: 'bob',
    school: 'No.1 middle school',
    address: 'xueyuan road'
};

var upper = _.map(obj, function (value, key) {
    return ???;
});

alert(JSON.stringify(upper));
```

你也许会想，为啥对Object作 `map()` 操作的返回结果是 `Array` ？应该是Object才合理啊！把 `_.map` 换成 `_.mapObject` 再试试。

every / some

当集合的所有元素都满足条件时， `_.every()` 函数返回 `true` ，当集合的至少一个元素满足条件时， `_.some()` 函数返回 `true` ：

```
'use strict';
// 所有元素都大于0？
_.every([1, 4, 7, -3, -9], (x) => x > 0); // false
// 至少一个元素大于0？
_.some([1, 4, 7, -3, -9], (x) => x > 0); // true
```

当集合是Object时，我们可以同时获得value和key：

```
'use strict';
var obj = {
    name: 'bob',
    school: 'No.1 middle school',
    address: 'xueyuan road'
};
// 判断key和value是否全部是小写：

var r1 = _.every(obj, function (value, key) {
    return ???;
});
var r2 = _.some(obj, function (value, key) {
    return ???;
});

alert('every key-value are lowercase: ' + r1 + '\nsome key-value ar
```

max / min

这两个函数直接返回集合中最大和最小的数：

```
'use strict';
var arr = [3, 5, 7, 9];
_.max(arr); // 9
_.min(arr); // 3

// 空集合会返回-Infinity和Infinity，所以要先判断集合不为空：
_.max([])
-Infinity
_.min([])
Infinity
```

注意，如果集合是Object， `max()` 和 `min()` 只作用于value，忽略掉key：

```
'use strict';
_.max({ a: 1, b: 2, c: 3 }); // 3
```

groupBy

`groupBy()` 把集合的元素按照key归类，key由传入的函数返回：

```
'use strict';

var scores = [20, 81, 75, 40, 91, 59, 77, 66, 72, 88, 99];
var groups = _.groupBy(scores, function (x) {
    if (x < 60) {
        return 'C';
    } else if (x < 80) {
        return 'B';
    } else {
        return 'A';
    }
});
// 结果:
// {
//   A: [81, 91, 88, 99],
//   B: [75, 77, 66, 72],
//   C: [20, 40, 59]
// }
```

可见 `groupBy()` 用来分组是非常方便的。

shuffle / sample

`shuffle()` 用洗牌算法随机打乱一个集合：

```
'use strict';
// 注意每次结果都不一样：
_.shuffle([1, 2, 3, 4, 5, 6]); // [3, 5, 4, 6, 2, 1]
```

`sample()` 则是随机选择一个或多个元素：

```
'use strict';
// 注意每次结果都不一样：
// 随机选1个：
_.sample([1, 2, 3, 4, 5, 6]); // 2
// 随机选3个：
_.sample([1, 2, 3, 4, 5, 6], 3); // [6, 1, 4]
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#collections

Arrays

underscore为 `Array` 提供了许多工具类方法，可以更方便快捷地操作 `Array` 。

first / last

顾名思义，这两个函数分别取第一个和最后一个元素：

```
'use strict';
var arr = [2, 4, 6, 8];
_.first(arr); // 2
_.last(arr); // 8
```

flatten

`flatten()` 接收一个 `Array` ，无论这个 `Array` 里面嵌套了多少个 `Array` ， `flatten()` 最后都把它们变成一个一维数组：

```
'use strict';

_.flatten([1, [2], [3, [[4], [5]]]]); // [1, 2, 3, 4, 5]
```

zip / unzip

`zip()` 把两个或多个数组的所有元素按索引对齐，然后按索引合并成新数组。例如，你有一个 `Array` 保存了名字，另一个 `Array` 保存了分数，现在，要把名字和分数给对上，用 `zip()` 轻松实现：

```
'use strict';

var names = ['Adam', 'Lisa', 'Bart'];
var scores = [85, 92, 59];
_.zip(names, scores);
// [['Adam', 85], ['Lisa', 92], ['Bart', 59]]
```

`unzip()` 则是反过来：

```
'use strict';
var namesAndScores = [['Adam', 85], ['Lisa', 92], ['Bart', 59]];
_.unzip(namesAndScores);
// [['Adam', 'Lisa', 'Bart'], [85, 92, 59]]
```

object

有时候你会想，与其用 `zip()` ，为啥不把名字和分数直接对应成Object呢？别急， `object()` 函数就是干这个的：

```
'use strict';

var names = ['Adam', 'Lisa', 'Bart'];
var scores = [85, 92, 59];
_.object(names, scores);
// {Adam: 85, Lisa: 92, Bart: 59}
```

注意 `_.object()` 是一个函数，不是JavaScript的 `Object` 对象。

range

`range()` 让你快速生成一个序列，不再需要用 `for` 循环实现了：

```
'use strict';

// 从0开始小于10:
_.range(10); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

// 从1开始小于11：
_.range(1, 11); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// 从0开始小于30，步长5:
_.range(0, 30, 5); // [0, 5, 10, 15, 20, 25]

// 从0开始大于-10，步长-1:
_.range(0, -10, -1); // [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#arrays

Functions

因为underscore本来就是为了充分发挥JavaScript的函数式编程特性，所以也提供了大量JavaScript本身没有的高阶函数。

bind

`bind()` 有什么用？我们先看一个常见的错误用法：

```
'use strict';

console.log('Hello, world!');
// 输出'Hello, world!'

var log = console.log;
log('Hello, world!');
// Uncaught TypeError: Illegal invocation
```

如果你想用 `log()` 取代 `console.log()` ，按照上面的做法是不行的，因为直接调用 `log()` 传入的 `this` 指针是 `undefined` ，必须这么用：

```
'use strict';

var log = console.log;
// 调用call并传入console对象作为this:
log.call(console, 'Hello, world!')
// 输出Hello, world!
```

这样搞多麻烦！还不如直接用 `console.log()` 。但是， `bind()` 可以帮我们把 `console` 对象直接绑定在 `log()` 的 `this` 指针上，以后调用 `log()` 就可以直接正常调用了：

```
'use strict';

var log = _.bind(console.log, console);
log('Hello, world!');
// 输出Hello, world!
```

partial

`partial()` 就是为一个函数创建偏函数。偏函数是什么东东？看例子：

假设我们要计算x^y，这时只需要调用 `Math.pow(x, y)` 就可以了。

假设我们经常计算2^y，每次都写 `Math.pow(2, y)` 就比较麻烦，如果创建一个新的函数能直接这样写 `pow2N(y)` 就好了，这个新函数 `pow2N(y)` 就是根据 `Math.pow(x, y)` 创建出来的偏函数，它固定住了原函数的第一个参数（始终为2）：

```
'use strict';

var pow2N = _.partial(Math.pow, 2);
pow2N(3); // 8
pow2N(5); // 32
pow2N(10); // 1024
```

如果我们不想固定第一个参数，想固定第二个参数怎么办？比如，希望创建一个偏函数 `cube(x)` ，计算x³，可以用 `_` 作占位符，固定住第二个参数：

```
'use strict';

var cube = _.partial(Math.pow, _, 3);
cube(3); // 27
cube(5); // 125
cube(10); // 1000
```

可见，创建偏函数的目的是将原函数的某些参数固定住，可以降低新函数调用的难度。

memoize

如果一个函数调用开销很大，我们就可能希望能把结果缓存下来，以便后续调用时直接获得结果。举个例子，计算阶乘就比较耗时：

```
'use strict';

function factorial(n) {
    console.log('start calculate ' + n + '!...');
    var s = 1, i = n;
    while (i > 1) {
        s = s * i;
        i --;
    }
    console.log(n + '! = ' + s);
    return s;
}

factorial(10); // 3628800
// 注意控制台输出:
// start calculate 10!...
// 10! = 3628800
```

用 `memoize()` 就可以自动缓存函数计算的结果：

```
'use strict';

var factorial = _.memoize(function(n) {
    console.log('start calculate ' + n + '!...');
    var s = 1, i = n;
    while (i > 1) {
        s = s * i;
        i --;
    }
    console.log(n + '! = ' + s);
    return s;
});

// 第一次调用:
factorial(10); // 3628800
// 注意控制台输出:
// start calculate 10!...
// 10! = 3628800

// 第二次调用:
factorial(10); // 3628800
// 控制台没有输出
```

对于相同的调用，比如连续两次调用 `factorial(10)` ，第二次调用并没有计算，而是直接返回上次计算后缓存的结果。不过，当你计算 `factorial(9)` 的时候，仍然会重新计算。

可以对 `factorial()` 进行改进，让其递归调用：

```
'use strict';

var factorial = _.memoize(function(n) {
    console.log('start calculate ' + n + '!...');
    if (n < 2) {
        return 1;
    }
    return n * factorial(n - 1);
});

factorial(10); // 3628800
// 输出结果说明factorial(1)~factorial(10)都已经缓存了:
// start calculate 10!...
// start calculate 9!...
// start calculate 8!...
// start calculate 7!...
// start calculate 6!...
// start calculate 5!...
// start calculate 4!...
// start calculate 3!...
// start calculate 2!...
// start calculate 1!...

factorial(9); // 362880
// console无输出
```

once

顾名思义， `once()` 保证某个函数执行且仅执行一次。如果你有一个方法叫 `register()` ，用户在页面上点两个按钮的任何一个都可以执行的话，就可以用 `once()` 保证函数仅调用一次，无论用户点击多少次：

```
'use strict';

var register = _.once(function () {
    alert('Register ok!');
});

// 测试效果:
register();
register();
register();
```

delay

`delay()` 可以让一个函数延迟执行，效果和 `setTimeout()` 是一样的，但是代码明显简单了：

```
'use strict';

// 2秒后调用alert():
_.delay(alert, 2000);
```

如果要延迟调用的函数有参数，把参数也传进去：

```
'use strict';

var log = _.bind(console.log, console);
_.delay(log, 2000, 'Hello,', 'world!');
// 2秒后打印'Hello, world!':
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#functions

Objects

和 `Array` 类似，underscore也提供了大量针对Object的函数。

keys / allKeys

`keys()` 可以非常方便地返回一个object自身所有的key，但不包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
    this.name = name;
    this.age = age;
}

var xiaoming = new Student('小明', 20);
_.keys(xiaoming); // ['name', 'age']
```

`allKeys()` 除了object自身的key，还包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
    this.name = name;
    this.age = age;
}
Student.prototype.school = 'No.1 Middle School';
var xiaoming = new Student('小明', 20);
_.allKeys(xiaoming); // ['name', 'age', 'school']
```

values

和 `keys()` 类似， `values()` 返回object自身但不包含原型链继承的所有值：

```
'use strict';

var obj = {
    name: '小明',
    age: 20
};

_.values(obj); // ['小明', 20]
```

注意，没有 `allValues()` ，原因我也不知道。

mapObject

`mapObject()` 就是针对object的map版本：

```
'use strict';

var obj = { a: 1, b: 2, c: 3 };
// 注意传入的函数签名，value在前，key在后:
_.mapObject(obj, (v, k) => 100 + v); // { a: 101, b: 102, c: 103 }
```

invert

`invert()` 把object的每个key-value来个交换，key变成value，value变成key：

```
'use strict';

var obj = {
    Adam: 90,
    Lisa: 85,
    Bart: 59
};
_.invert(obj); // { '59': 'Bart', '85': 'Lisa', '90': 'Adam' }
```

extend / extendOwn

`extend()` 把多个object的key-value合并到第一个object并返回：

```
'use strict';

var a = {name: 'Bob', age: 20};
_.extend(a, {age: 15}, {age: 88, city: 'Beijing'}); // {name: 'Bob
// 变量a的内容也改变了：
a; // {name: 'Bob', age: 88, city: 'Beijing'}
```

注意：如果有相同的key，后面的object的value将覆盖前面的object的value。

`extendOwn()` 和 `extend()` 类似，但获取属性时忽略从原型链继承下来的属性。

clone

如果我们要复制一个object对象，就可以用 `clone()` 方法，它会把原有对象的所有属性都复制到新的对象中：

```
'use strict';
var source = {
    name: '小明',
    age: 20,
    skills: ['JavaScript', 'CSS', 'HTML']
};

var copied = _.clone(source);

alert(JSON.stringify(copied, null, '  '));
```

注意， `clone()` 是“浅复制”。所谓“浅复制”就是说，两个对象相同的key所引用的value其实是同一对象：

```
source.skills === copied.skills; // true
```

也就是说，修改 `source.skills` 会影响 `copied.skills` 。

isEqual

`isEqual()` 对两个object进行深度比较，如果内容完全相同，则返回 `true` ：

```
'use strict';

var o1 = { name: 'Bob', skills: { Java: 90, JavaScript: 99 }};
var o2 = { name: 'Bob', skills: { JavaScript: 99, Java: 90 }};

o1 === o2; // false
_.isEqual(o1, o2); // true
```

`isEqual()` 其实对 `Array` 也可以比较：

```
'use strict';

var o1 = ['Bob', { skills: ['Java', 'JavaScript'] }];
var o2 = ['Bob', { skills: ['Java', 'JavaScript'] }];

o1 === o2; // false
_.isEqual(o1, o2); // true
```

更多完整的函数请参考underscore的文档：http://underscorejs.org/#objects

Chaining

还记得jQuery支持链式调用吗？

```
$('a').attr('target', '_blank')
      .append(' <i class="uk-icon-external-link"></i>')
      .click(function () {});
```

如果我们有一组操作，用underscore提供的函数，写出来像这样：

```
_.filter(_.map([1, 4, 9, 16, 25], Math.sqrt), x => x % 2 === 1);
// [1, 3, 5]
```

能不能写成链式调用？

能！

underscore提供了把对象包装成能进行链式调用的方法，就是 `chain()` 函数：

```
_.chain([1, 4, 9, 16, 25])
 .map(Math.sqrt)
 .filter(x => x % 2 === 1)
 .value();
// [1, 3, 5]
```

因为每一步返回的都是包装对象，所以最后一步的结果需要调用 `value()` 获得最终结果。

小结

通过学习underscore，是不是对JavaScript的函数式编程又有了进一步的认识？

Node.js

从本章开始，我们就正式开启JavaScript的后端开发之旅。

Node.js是目前非常火热的技术，但是它的诞生经历却很奇特。

众所周知，在Netscape设计出JavaScript后的短短几个月，JavaScript事实上已经是前端开发的唯一标准。

后来，微软通过IE击败了Netscape后一统桌面，结果几年时间，浏览器毫无进步。（2001年推出的古老的IE 6到今天仍然有人在使用！）

没有竞争就没有发展。微软认为IE6浏览器已经非常完善，几乎没有可改进之处，然后解散了IE6开发团队！而Google却认为支持现代Web应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行JavaScript的引擎性能还可提升10倍。

先是Mozilla借助已壮烈牺牲的Netscape遗产在2002年推出了Firefox浏览器，紧接着Apple于2003年在开源的KHTML浏览器的基础上推出了WebKit内核的Safari浏览器，不过仅限于Mac平台。

随后，Google也开始创建自家的浏览器。他们也看中了WebKit内核，于是基于WebKit内核推出了Chrome浏览器。

Chrome浏览器是跨Windows和Mac平台的，并且，Google认为要运行现代Web应用，浏览器必须有一个性能非常强劲的JavaScript引擎，于是Google自己开发了一个高性能JavaScript引擎，名字叫V8，以BSD许可证开源。

现代浏览器大战让微软的IE浏览器远远地落后了，因为他们解散了最有经验、战斗力最强的浏览器团队！回过头再追赶却发现，支持HTML5的WebKit已经成为手机端的标准了，IE浏览器从此与主流移动端设备绝缘。

浏览器大战和Node有何关系？

话说有个叫Ryan Dahl的歪果仁，他的工作是用C/C++写高性能Web服务。对于高性能，异步IO、事件驱动是基本原则，但是用C/C++写就太痛苦了。于是这位仁兄开始设想用高级语言开发Web服务。他评估了很多种高级语言，发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，他们就再也懒得写异步IO了，所以，最终，Ryan瞄向了JavaScript。

因为JavaScript是单线程执行，根本不能进行同步IO操作，所以，JavaScript的这一“缺陷”导致了它只能使用异步IO。

选定了开发语言，还要有运行时引擎。这位仁兄曾考虑过自己写一个，不过明智地放弃了，因为V8就是开源的JavaScript引擎。让Google投资去优化V8，咱只负责改造一下拿来用，还不用付钱，这个买卖很划算。

于是在2009年，Ryan正式推出了基于JavaScript语言和V8引擎的开源Web服务器项目，命名为Node.js。虽然名字很土，但是，Node第一次把JavaScript带入到后端服务器开发，加上世界上已经有无数的JavaScript开发人员，所以Node一下子就火了起来。

在Node上运行的JavaScript相比其他后端开发语言有何优势？

最大的优势是借助JavaScript天生的事件驱动机制加V8高性能引擎，使编写高性能Web服务轻而易举。

其次，JavaScript语言本身是完善的函数式语言，在前端开发时，开发人员往往写得比较随意，让人感觉JavaScript就是个“玩具语言”。但是，在Node环境下，通过模块化的JavaScript代码，加上函数式编程，并且无需考虑浏览器兼容性问题，直接使用最新的ECMAScript 6标准，可以完全满足工程上的需求。

> 我还听说过io.js，这又是什么鬼？

因为Node.js是开源项目，虽然由社区推动，但幕后一直由Joyent公司资助。由于一群开发者对Joyent公司的策略不满，于2014年从Node.js项目fork出了io.js项目，决定单独发展，但两者实际上是兼容的。

然而中国有句古话，叫做“分久必合，合久必分”。分家后没多久，Joyent公司表示要和解，于是，io.js项目又决定回归Node.js。

具体做法是将来io.js将首先添加新的特性，如果大家测试用得爽，就把新特性加入Node.js。io.js是“尝鲜版”，而Node.js是线上稳定版，相当于Fedora Linux和RHEL的关系。

本章教程的所有代码都在Node.js上调试通过。如果你要尝试io.js也是可以的，不过两者如果遇到一些区别请自行查看io.js的文档。

安装Node.js和npm

由于Node.js平台是在后端运行JavaScript代码，所以，必须首先在本机安装Node环境。

安装Node.js

目前Node.js的最新版本是5.3.x。首先，从Node.js官网下载对应平台的安装程序，网速慢的童鞋请移步国内镜像。

在Windows上安装时务必选择全部组件，包括勾选 `Add to Path` 。

安装完成后，在Windows环境下，请打开命令提示符，然后输入 `node -v` ，如果安装正常，你应该看到 `v5.3.0` 这样的输出：

```
C:\Users\IEUser>node -v
v5.3.0
```

继续在命令提示符输入 `node` ，此刻你将进入Node.js的交互环境。在交互环境下，你可以输入任意JavaScript语句，例如 `100+200` ，回车后将得到输出结果。

要退出Node.js环境，连按两次Ctrl+C。

http://michaelliao.gitcafe.io/video/node/install-node.mp4

在Mac或Linux环境下，请打开终端，然后输入 `node -v` ，你应该看到如下输出：

```
$ node -v
v5.3.0
```

如果版本号不是 `5.3.x` ，说明Node.js版本不对，后面章节的代码不保证能正常运行，请重新安装最新版本。

npm

在正式开始Node.js学习之前，我们先认识一下npm。

npm是什么东东？npm其实是Node.js的包管理工具（package manager）。

为啥我们需要一个包管理工具呢？因为我们在Node.js上开发时，会用到很多别人写的JavaScript代码。如果我们要使用别人写的某个包，每次都根据名称搜索一下官方网站，下载代码，解压，再使用，非常繁琐。于是一个集中管理的工具应运而生：大家都把自己开发的模块打包后放到npm官网上，如果要使用，直接通过npm安装就可以直接用，不用管代码存在哪，应该从哪下载。

更重要的是，如果我们要使用模块A，而模块A又依赖于模块B，模块B又依赖于模块X和模块Y，npm可以根据依赖关系，把所有依赖的包都下载下来并管理起来。否则，靠我们自己手动管理，肯定既麻烦又容易出错。

讲了这么多，npm究竟在哪？

其实npm已经在Node.js安装的时候顺带装好了。我们在命令提示符或者终端输入 `npm -v` ，应该看到类似的输出：

```
C:\>npm -v
3.3.12
```

如果直接输入 `npm` ，你会看到类似下面的输出：

```
C:\> npm

Usage: npm <command>

where <command> is one of:
    ...
```

上面的一大堆文字告诉你， `npm` 需要跟上命令。现在我们不用关心这些命令，后面会一一讲到。目前，你只需要确保npm正确安装了，能运行就行。

小结

请在本机安装Node.js环境，并确保 `node` 和 `npm` 能正常运行。

第一个Node程序

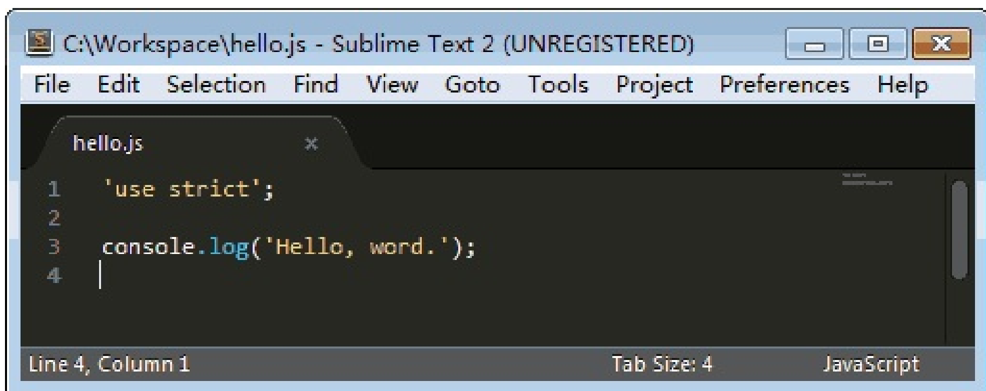
在前面的所有章节中，我们编写的JavaScript代码都是在浏览器中运行的，因此，我们可以直接在浏览器中敲代码，然后直接运行。

从本章开始，我们编写的JavaScript代码将不能在浏览器环境中执行了，而是在Node环境中执行，因此，JavaScript代码将直接在你的计算机上以命令行的方式运行，所以，我们要先选择一个文本编辑器来编写JavaScript代码，并且把它保存到本地硬盘的某个目录，才能够执行。

那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是Sublime Text，免费使用，但是不付费会弹出提示框：



一个是Notepad++，免费使用，有中文界面：



请注意，用哪个都行，但是绝对不能用Word和写字板，Windows自带的记事本也强烈不推荐使用。Word和写字板保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果经常会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

```
'use strict';

console.log('Hello, world.');
```

第一行总是写上 `'use strict';` 是因为我们总是以严格模式运行JavaScript代码，避免各种潜在陷阱。

然后，选择一个目录，例如 `C:\Workspace` ，把文件保存为 `hello.js` ，就可以打开命令行窗口，把当前目录切换到 `hello.js` 所在目录，然后输入以下命令运行这个程序了：

```
C:\Workspace>node hello.js
Hello, world.
```

也可以保存为别的名字，比如 `first.js` ，但是必须要以 `.js` 结尾。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有 `hello.js` 这个文件，运行 `node hello.js` 就会报错：

```
C:\Workspace>node hello.js
module.js:338
    throw err;
          ^
Error: Cannot find module 'C:\Workspace\hello.js'
    at Function.Module._resolveFilename
    at Function.Module._load
    at Function.Module.runMain
    at startup
    at node.js
```

报错的意思就是，没有找到 `hello.js` 这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

命令行模式和**Node**交互模式

请注意区分命令行模式和Node交互模式。

看到类似 `C:\>` 是在Windows提供的命令行模式：



在命令行模式下，可以执行 `node` 进入Node交互式环境，也可以执行 `node hello.js` 运行一个 `.js` 文件。

看到 `>` 是在Node交互式环境下：



在Node交互式环境下，我们可以输入JavaScript代码并立刻执行。

此外，在命令行模式运行 `.js` 文件和在Node交互式环境下直接运行JavaScript代码有所不同。Node交互式环境会把每一行JavaScript代码的结果自动打印出来，但是，直接运行JavaScript文件却不会。

例如，在Node交互式环境下，输入：

```
> 100 + 200 + 300;
600
```

直接可以看到结果 `600` 。

但是，写一个 `calc.js` 的文件，内容如下：

```
100 + 200 + 300;
```

然后在命令行模式下执行：

```
C:\Workspace>node calc.js
```

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用 `console.log()` 打印出来。把 `calc.js` 改造一下：

```
console.log(100 + 200 + 300);
```

再执行，就可以看到结果：

```
C:\Workspace>node calc.js
600
```

小结

用文本编辑器写JavaScript程序，然后保存为后缀为 `.js` 的文件，就可以用node直接运行这个程序了。

Node的交互模式和直接运行 `.js` 文件有什么区别呢？

直接输入 `node` 进入交互模式，相当于启动了Node解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行 `node hello.js` 文件相当于启动了Node解释器，然后一次性把 `hello.js` 文件的源代码给执行了，你是没有机会以交互的方式输入源代码的。

在编写JavaScript代码的时候，完全可以一边在文本编辑器里写代码，一边开一个Node交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

参考源码

hello.js

模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Node环境中，一个.js文件就称之为一个模块（module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Node内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。

在上一节，我们编写了一个 `hello.js` 文件，这个 `hello.js` 文件就是一个模块，模块的名字就是文件名（去掉 `.js` 后缀），所以 `hello.js` 文件就是名为 `hello` 的模块。

我们把 `hello.js` 改造一下，创建一个函数，这样我们就可以在其他地方调用这个函数：

```
'use strict';

var s = 'Hello';

function greet(name) {
    console.log(s + ', ' + name + '!');
}

module.exports = greet;
```

函数 `greet()` 是我们在 `hello` 模块中定义的，你可能注意到最后一行是一个奇怪的赋值语句，它的意思是，把函数 `greet` 作为模块的输出暴露出去，这样其他模块就可以使用 `greet` 函数了。

问题是其他模块怎么使用 `hello` 模块的这个 `greet` 函数呢？我们再编写一个 `main.js` 文件，调用 `hello` 模块的 `greet` 函数：

```
'use strict';

// 引入hello模块:
var greet = require('./hello');

var s = 'Michael';

greet(s); // Hello, Michael!
```

注意到引入 `hello` 模块用Node提供的 `require` 函数：

```
var greet = require('./hello');
```

引入的模块作为变量保存在 `greet` 变量中，那 `greet` 变量到底是什么东西？其实变量 `greet` 就是在 `hello.js` 中我们用 `module.exports = greet;` 输出的 `greet` 函数。所以， `main.js` 就成功地引用了 `hello.js` 模块中定义的 `greet()` 函数，接下来就可以直接使用它了。

在使用 `require()` 引入模块的时候，请注意模块的相对路径。因为 `main.js` 和 `hello.js` 位于同一个目录，所以我们用了当前目录 `.` ：

```
var greet = require('./hello'); // 不要忘了写相对目录!
```

如果只写模块名：

```
var greet = require('hello');
```

则Node会依次在内置模块、全局模块和当前模块下查找 `hello.js` ，你很可能会得到一个错误：

```
module.js
    throw err;
          ^
Error: Cannot find module 'hello'
    at Function.Module._resolveFilename
    at Function.Module._load
    ...
    at Function.Module._load
    at Function.Module.runMain
```

遇到这个错误，你要检查：

- 模块名是否写对了；
- 模块文件是否存在；
- 相对路径是否写对了。

CommonJS规范

这种模块加载机制被称为CommonJS规范。在这个规范下，每个 `.js` 文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突，例如， `hello.js` 和 `main.js` 都申明了全局变量 `var s = 'xxx'` ，但互不影响。

一个模块想要对外暴露变量（函数也是变量），可以用 `module.exports = variable;` ，一个模块要引用其他模块暴露的变量，用 `var ref = require('module_name');` 就拿到了引用模块的变量。

结论

要在模块中对外输出变量，用：

```
module.exports = variable;
```

输出的变量可以是任意对象、函数、数组等等。

要引入其他模块输出的对象，用：

```
var foo = require('other_module');
```

引入的对象具体是什么，取决于引入模块输出的对象。

深入了解模块原理

如果你想详细地了解CommonJS的模块实现原理，请继续往下阅读。如果不想了解，请直接跳到最后做练习。

当我们编写JavaScript代码时，我们可以申明全局变量：

```
var s = 'global';
```

在浏览器中，大量使用全局变量可不好。如果你在 `a.js` 中使用了全局变量 `s` ，那么，在 `b.js` 中也使用全局变量 `s` ，将造成冲突， `b.js` 中对 `s` 赋值会改变 `a.js` 的运行逻辑。

也就是说，JavaScript语言本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

那Node.js是如何实现这一点的？

其实要实现“模块”这个功能，并不需要语法层面的支持。Node.js也并不会增加任何JavaScript语法。实现“模块”功能的奥妙就在于JavaScript是一种函数式编程语言，它支持闭包。如果我们把一段JavaScript代码用一个函数包装起来，这段代码的所有“全局”变量就变成了函数内部的局部变量。

请注意我们编写的 `hello.js` 代码是这样的：

```
var s = 'Hello';
var name = 'world';

console.log(s + ' ' + name + '!');
```

Node.js加载了 `hello.js` 后，它可以把代码包装一下，变成这样执行：

```
(function () {
    // 读取的hello.js代码:
    var s = 'Hello';
    var name = 'world';

    console.log(s + ' ' + name + '!');
    // hello.js代码结束
})();
```

这样一来，原来的全局变量 `s` 现在变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的“全局”变量 `s` 也互不干扰。

所以，Node利用JavaScript的函数式编程的特性，轻而易举地实现了模块的隔离。

但是，模块的输出 `module.exports` 怎么实现？

这个也很容易实现，Node可以先准备一个对象 `module` ：

```
// 准备module对象:
var module = {
    id: 'hello',
    exports: {}
};
var load = function (module) {
    // 读取的hello.js代码:
    function greet(name) {
        console.log('Hello, ' + name + '!');
    }

    module.exports = greet;
    // hello.js代码结束
    return module.exports;
};
var exported = load(module);
// 保存module:
save(module, exported);
```

可见，变量 `module` 是Node在加载js文件前准备的一个变量，并将其传入加载函数，我们在 `hello.js` 中可以直接使用变量 `module` 原因就在于它实际上是函数的一个参数：

```
module.exports = greet;
```

通过把参数 `module` 传递给 `load()` 函数， `hello.js` 就顺利地把一个变量传递给了Node执行环境，Node会把 `module` 变量保存到某个地方。

由于Node保存了所有导入的 `module` ，当我们用 `require()` 获取module时，Node找到对应的 `module` ，把这个 `module` 的 `exports` 变量返回，这样，另一个模块就顺利拿到了模块的输出：

```
var greet = require('./hello');
```

以上是Node实现JavaScript模块的一个简单的原理介绍。

module.exports vs exports

很多时候，你会看到，在Node环境中，有两种方法可以在一个模块中输出变量：

方法一：对module.exports赋值：

```
// hello.js

function hello() {
    console.log('Hello, world!');
}

function greet(name) {
    console.log('Hello, ' + name + '!');
}

function hello() {
    console.log('Hello, world!');
}

module.exports = {
    hello: hello,
    greet: greet
};
```

方法二：直接使用exports：

```
// hello.js

function hello() {
    console.log('Hello, world!');
}

function greet(name) {
    console.log('Hello, ' + name + '!');
}

function hello() {
    console.log('Hello, world!');
}

exports.hello = hello;
exports.greet = greet;
```

但是你不可以直接对 `exports` 赋值：

```
// 代码可以执行，但是模块并没有输出任何变量:
exports = {
    hello: hello,
    greet: greet
};
```

如果你对上面的写法感到十分困惑，不要着急，我们来分析Node的加载机制：

首先，Node会把整个待加载的 `hello.js` 文件放入一个包装函数 `load` 中执行。在执行这个 `load()` 函数前，Node准备好了module变量：

```
var module = {
    id: 'hello',
    exports: {}
};
```

`load()` 函数最终返回 `module.exports` ：

```
var load = function (exports, module) {
    // hello.js的文件内容
    ...
    // load函数返回:
    return module.exports;
};

var exported = load(module.exports, module);
```

也就是说，默认情况下，Node准备的 `exports` 变量和 `module.exports` 变量实际上是同一个变量，并且初始化为空对象 `{}` ，于是，我们可以写：

```
exports.foo = function () { return 'foo'; };
exports.bar = function () { return 'bar'; };
```

也可以写：

```
module.exports.foo = function () { return 'foo'; };
module.exports.bar = function () { return 'bar'; };
```

换句话说，Node默认给你准备了一个空对象 `{}` ，这样你可以直接往里面加东西。

但是，如果我们要输出的是一个函数或数组，那么，只能给 `module.exports` 赋值：

```
module.exports = function () { return 'foo'; };
```

给 `exports` 赋值是无效的，因为赋值后， `module.exports` 仍然是空对象 `{}` 。

结论

如果要输出一个键值对象 `{}` ，可以利用 `exports` 这个已存在的空对象 `{}` ，并继续在上面添加新的键值；

如果要输出一个函数或数组，必须直接对 `module.exports` 对象赋值。

所以我们可以得出结论：直接对 `module.exports` 赋值，可以应对任何情况：

```
module.exports = {
    foo: function () { return 'foo'; }
};
```

或者：

```
module.exports = function () { return 'foo'; };
```

最终，我们强烈建议使用 `module.exports = xxx` 的方式来输出模块变量，这样，你只需要记忆一种方法。

练习

编写 `hello.js` ，输出一个或多个函数；

编写 `main.js` ，引入 `hello` 模块，调用其函数。

参考源码

hello.js

main.js

基本模块

因为Node.js是运行在服务区端的JavaScript环境，服务器程序和浏览器程序相比，最大的特点是没有浏览器的安全限制了，而且，服务器程序必须能接收网络请求，读写文件，处理二进制内容，所以，Node.js内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中是无法被执行的，因为它们的底层代码是用C/C++在Node.js运行环境中实现的。

global

在前面的JavaScript课程中，我们已经知道，JavaScript有且仅有一个全局对象，在浏览器中，叫 `window` 对象。而在Node.js环境中，也有唯一的全局对象，但不叫 `window` ，而叫 `global` ，这个对象的属性和方法也和浏览器环境的 `window` 不同。进入Node.js交互环境，可以直接输入：

```
> global.console
Console {
  log: [Function: bound ],
  info: [Function: bound ],
  warn: [Function: bound ],
  error: [Function: bound ],
  dir: [Function: bound ],
  time: [Function: bound ],
  timeEnd: [Function: bound ],
  trace: [Function: bound trace],
  assert: [Function: bound ],
  Console: [Function: Console] }
```

process

`process` 也是Node.js提供的一个对象，它代表当前Node.js进程。通过 `process` 对象可以拿到许多有用信息：

```
> process === global.process;
true
> process.version;
'v5.2.0'
> process.platform;
'darwin'
> process.arch;
'x64'
> process.cwd(); //返回当前工作目录
'/Users/michael'
> process.chdir('/private/tmp'); // 切换当前工作目录
undefined
> process.cwd();
'/private/tmp'
```

JavaScript程序是由事件驱动执行的单线程模型，Node.js也不例外。Node.js不断执行响应事件的JavaScript函数，直到没有任何响应事件的函数可以执行时，Node.js就退出了。

如果我们想要在下一次事件响应中执行代码，可以调用 `process.nextTick()` ：

```
// test.js

// process.nextTick()将在下一轮事件循环中调用:
process.nextTick(function () {
    console.log('nextTick callback!');
});
console.log('nextTick was set!');
```

用Node执行上面的代码 `node test.js` ，你会看到，打印输出是：

```
nextTick was set!
nextTick callback!
```

这说明传入 `process.nextTick()` 的函数不是立刻执行，而是要等到下一次事件循环。

Node.js进程本身的事件就由 `process` 对象来处理。如果我们响应 `exit` 事件，就可以在程序即将退出时执行某个回调函数：

```
// 程序即将退出时的回调函数:
process.on('exit', function (code) {
    console.log('about to exit with code: ' + code);
});
```

判断JavaScript执行环境

有很多JavaScript代码既能在浏览器中执行，也能在Node环境执行，但有些时候，程序本身需要判断自己到底是在什么环境下执行的，常用的方式就是根据浏览器和Node环境提供的全局变量名称来判断：

```
if (typeof(window) === 'undefined') {
    console.log('node.js');
} else {
    console.log('browser');
}
```

后面，我们将介绍Node.js的常用内置模块。

参考源码

gl.js

fs

Node.js内置的 `fs` 模块就是文件系统模块，负责读写文件。

和所有其它JavaScript模块不同的是， `fs` 模块同时提供了异步和同步的方法。

回顾一下什么是异步方法。因为JavaScript的单线程模型，执行IO操作时，JavaScript代码无需等待，而是传入回调函数后，继续执行后续JavaScript代码。比如jQuery提供的 `getJSON()` 操作：

```
$.getJSON('http://example.com/ajax', function (data) {
    console.log('IO结果返回后执行...');
});
console.log('不等待IO结果直接执行后续代码...');
```

而同步的IO操作则需要等待函数返回：

```
// 根据网络耗时，函数将执行几十毫秒~几秒不等:
var data = getJSONSync('http://example.com/ajax');
```

同步操作的好处是代码简单，缺点是程序将等待IO操作，在等待时间内，无法响应其它任何事件。而异步读取不用等待IO操作，但代码较麻烦。

异步读文件

按照JavaScript的标准，异步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.txt', 'utf-8', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
```

请注意，`sample.txt`文件必须在当前目录下，且文件编码为`utf-8`。

异步读取时，传入的回调函数接收两个参数，当正常读取时，`err`参数为`null`，`data`参数为读取到的String。当读取发生错误时，`err`参数代表一个错误对象，`data`为`undefined`。这也是Node.js标准的回调函数：第一个参数代表错误信息，第二个参数代表结果。后面我们还会经常编写这种回调函数。

由于`err`是否为`null`就是判断是否出错的标志，所以通常的判断逻辑总是：

```
if (err) {
    // 出错了
} else {
    // 正常
}
```

如果我们要读取的文件不是文本文件，而是二进制文件，怎么办？

下面的例子演示了如何读取一个图片文件：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.png', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
        console.log(data.length + ' bytes');
    }
});
```

当读取二进制文件时，不传入文件编码时，回调函数的 `data` 参数将返回一个 `Buffer` 对象。在Node.js中， `Buffer` 对象就是一个包含零个或任意个字节的数组（注意和Array不同）。

`Buffer` 对象可以和String作转换，例如，把一个 `Buffer` 对象转换成String：

```
// Buffer -> String
var text = data.toString('utf-8');
console.log(text);
```

或者把一个String转换成 `Buffer` ：

```
// String -> Buffer
var buf = new Buffer(text, 'utf-8');
console.log(buf);
```

同步读文件

除了标准的异步读取模式外， `fs` 也提供相应的同步读取函数。同步读取的函数和异步函数相比，多了一个 `Sync` 后缀，并且不接收回调函数，函数直接返回结果。

用 `fs` 模块同步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

var data = fs.readFileSync('sample.txt', 'utf-8');
console.log(data);
```

可见，原异步调用的回调函数的 `data` 被函数直接返回，函数名需要改为 `readFileSync` ，其它参数不变。

如果同步读取文件发生错误，则需要用 `try...catch` 捕获该错误：

```
try {
    var data = fs.readFileSync('sample.txt', 'utf-8');
    console.log(data);
} catch (err) {
    // 出错了
}
```

写文件

将数据写入文件是通过 `fs.writeFile()` 实现的：

```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFile('output.txt', data, function (err) {
    if (err) {
        console.log(err);
    } else {
        console.log('ok.');
    }
});
```

`writeFile()` 的参数依次为文件名、数据和回调函数。如果传入的数据是String，默认按UTF-8编码写入文本文件，如果传入的参数是 `Buffer` ，则写入的是二进制文件。回调函数由于只关心成功与否，因此只需要一个 `err` 参数。

和 `readFile()` 类似， `writeFile()` 也有一个同步方法，叫 `writeFileSync()` ：

```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFileSync('output.txt', data);
```

stat

如果我们要获取文件大小，创建时间等信息，可以使用 `fs.stat()` ，它返回一个 `Stat` 对象，能告诉我们文件或目录的详细信息：

```
'use strict';

var fs = require('fs');

fs.stat('sample.txt', function (err, stat) {
    if (err) {
        console.log(err);
    } else {
        // 是否是文件:
        console.log('isFile: ' + stat.isFile());
        // 是否是目录:
        console.log('isDirectory: ' + stat.isDirectory());
        if (stat.isFile()) {
            // 文件大小:
            console.log('size: ' + stat.size);
            // 创建时间, Date对象:
            console.log('birth time: ' + stat.birthtime);
            // 修改时间, Date对象:
            console.log('modified time: ' + stat.mtime);
        }
    }
});
```

运行结果如下：

```
isFile: true
isDirectory: false
size: 181
birth time: Fri Dec 11 2015 09:43:41 GMT+0800 (CST)
modified time: Fri Dec 11 2015 12:09:00 GMT+0800 (CST)
```

`stat()` 也有一个对应的同步函数 `statSync()` ，请试着改写上述异步代码为同步代码。

异步还是同步

在 `fs` 模块中，提供同步方法是为了方便使用。那我们到底是应该用异步方法还是同步方法呢？

由于Node环境执行的JavaScript代码是服务器端代码，所以，绝大部分需要在服务器运行期反复执行业务逻辑的代码，必须使用异步代码，否则，同步代码在执行时期，服务器将停止响应，因为JavaScript只有一个执行线程。

服务器启动时如果需要读取配置文件，或者结束时需要写入到状态文件时，可以使用同步代码，因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

参考源码

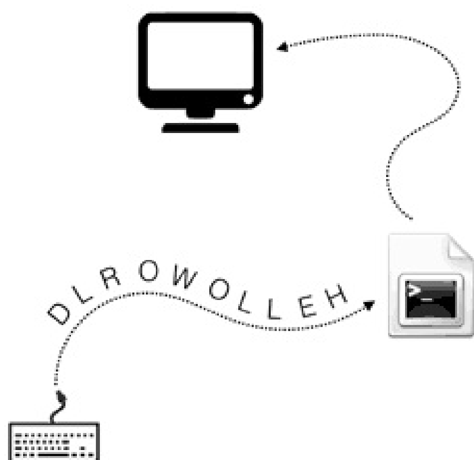
用fs读写文件

stream

`stream` 是Node.js提供的又一个仅在服务区端可用的模块，目的是支持“流”这种数据结构。

什么是流？流是一种抽象的数据结构。想象水流，当在水管中流动时，就可以从某个地方（例如自来水厂）源源不断地到达另一个地方（比如你家的洗手池）。我们也可以把数据看成是数据流，比如你敲键盘的时候，就可以把每个字符依次连起来，看成字符流。这个流是从键盘输入到应用程序，实际上它还对应着一个名字：标准输入流（stdin）。

如果应用程序把字符一个一个输出到显示器上，这也可以看成是一个流，这个流也有名字：标准输出流（stdout）。流的特点是数据是有序的，而且必须依次读取，或者依次写入，不能像Array那样随机定位。



有些流用来读取数据，比如从文件读取数据时，可以打开一个文件流，然后从文件流中不断地读取数据。有些流用来写入数据，比如向文件写入数据时，只需要把数据不断地往文件流中写进去就可以了。

在Node.js中，流也是一个对象，我们只需要响应流的事件就可以了： `data` 事件表示流的数据已经可以读取了， `end` 事件表示这个流已经到末尾了，没有数据可以读取了， `error` 事件表示出错了。

下面是一个从文件流读取文本内容的示例：

```
'use strict';

var fs = require('fs');

// 打开一个流:
var rs = fs.createReadStream('sample.txt', 'utf-8');

rs.on('data', function (chunk) {
    console.log('DATA:')
    console.log(chunk);
});

rs.on('end', function () {
    console.log('END');
});

rs.on('error', function (err) {
    console.log('ERROR: ' + err);
});
```

要注意，`data` 事件可能会有多次，每次传递的 `chunk` 是流的一部分数据。

要以流的形式写入文件，只需要不断调用 `write()` 方法，最后以 `end()` 结束：

```
'use strict';

var fs = require('fs');

var ws1 = fs.createWriteStream('output1.txt', 'utf-8');
ws1.write('使用Stream写入文本数据...\n');
ws1.write('END.');
ws1.end();

var ws2 = fs.createWriteStream('output2.txt');
ws2.write(new Buffer('使用Stream写入二进制数据...\n', 'utf-8'));
ws2.write(new Buffer('END.', 'utf-8'));
ws2.end();
```

所有可以读取数据的流都继承自 `stream.Readable` ，所有可以写入的流都继承自 `stream.Writable` 。

pipe

就像可以把两个水管串成一个更长的水管一样，两个流也可以串起来。一个 `Readable` 流和一个 `Writable` 流串起来后，所有的数据自动从 `Readable` 流进入 `Writable` 流，这种操作叫 `pipe` 。

在Node.js中， `Readable` 流有一个 `pipe()` 方法，就是用来干这件事的。

让我们用 `pipe()` 把一个文件流和另一个文件流串起来，这样源文件的所有数据就自动写入到目标文件里了，所以，这实际上是一个复制文件的程序：

```
'use strict';

var fs = require('fs');

var rs = fs.createReadStream('sample.txt');
var ws = fs.createWriteStream('copied.txt');

rs.pipe(ws);
```

默认情况下，当 `Readable` 流的数据读取完毕， `end` 事件触发后，将自动关闭 `Writable` 流。如果我们不希望自动关闭 `Writable` 流，需要传入参数：

```
readable.pipe(writable, { end: false });
```

参考源码

stream

http

Node.js开发的目的就是为了用JavaScript编写Web服务器程序。因为JavaScript实际上已经统治了浏览器端的脚本，其优势就是有世界上数量最多的前端开发人员。如果已经掌握了JavaScript前端开发，再学习一下如何将JavaScript应用在后端开发，就是名副其实的全栈了。

HTTP协议

要理解Web服务器程序的工作原理，首先，我们要对HTTP协议有基本的了解。如果你对HTTP协议不太熟悉，先看一看HTTP协议简介。

HTTP服务器

要开发HTTP服务器程序，从头处理TCP连接，解析HTTP是不现实的。这些工作实际上已经由Node.js自带的 `http` 模块完成了。应用程序并不直接和HTTP协议打交道，而是操作 `http` 模块提供的 `request` 和 `response` 对象。

`request` 对象封装了HTTP请求，我们调用 `request` 对象的属性和方法就可以拿到所有HTTP请求的信息；

`response` 对象封装了HTTP响应，我们操作 `response` 对象的方法，就可以把HTTP响应返回给浏览器。

用Node.js实现一个HTTP服务器程序非常简单。我们来实现一个最简单的Web程序 `hello.js` ，它对于所有请求，都返回 `Hello world!` ：

```
'use strict';

// 导入http模块:
var http = require('http');

// 创建http server，并传入回调函数:
var server = http.createServer(function (request, response) {
    // 回调函数接收request和response对象,
    // 获得HTTP请求的method和url:
    console.log(request.method + ': ' + request.url);
    // 将HTTP响应200写入response, 同时设置Content-Type: text/html:
    response.writeHead(200, {'Content-Type': 'text/html'});
    // 将HTTP响应的HTML内容写入response:
    response.end('<h1>Hello world!</h1>');
});

// 让服务器监听8080端口:
server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');
```

在命令提示符下运行该程序，可以看到以下输出：

```
$ node hello.js
Server is running at http://127.0.0.1:8080/
```

不要关闭命令提示符，直接打开浏览器输入 `http://localhost:8080` ，即可看到服务器响应的内容：

同时，在命令提示符窗口，可以看到程序打印的请求信息：

```
GET: /
GET: /favicon.ico
```

这就是我们编写的第一个HTTP服务器程序！

文件服务器

让我们继续扩展一下上面的Web程序。我们可以设定一个目录，然后让Web程序变成一个文件服务器。要实现这一点，我们只需要解析 `request.url` 中的路径，然后在本地找到对应的文件，把文件内容发送出去就可以了。

解析URL需要用到Node.js提供的 `url` 模块，它使用起来非常简单，通过 `parse()` 将一个字符串解析为一个 `Url` 对象：

```
'use strict';

var url = require('url');

console.log(url.parse('http://user:pass@host.com:8080/path/to/file?
```

结果如下：

```
Url {
  protocol: 'http:',
  slashes: true,
  auth: 'user:pass',
  host: 'host.com:8080',
  port: '8080',
  hostname: 'host.com',
  hash: '#hash',
  search: '?query=string',
  query: 'query=string',
  pathname: '/path/to/file',
  path: '/path/to/file?query=string',
  href: 'http://user:pass@host.com:8080/path/to/file?query=string#h
```

处理本地文件目录需要使用Node.js提供的 `path` 模块，它可以方便地构造目录：

```
'use strict';

var path = require('path');

// 解析当前目录:
var workDir = path.resolve('.'); // '/Users/michael'

// 组合完整的文件路径:当前目录+'pub'+'index.html':
var filePath = path.join(workDir, 'pub', 'index.html');
// '/Users/michael/pub/index.html'
```

使用 `path` 模块可以正确处理操作系统相关的文件路径。在Windows系统下，返回的路径类似于 `C:\Users\michael\static\index.html` ，这样，我们就不关心怎么拼接路径了。

最后，我们实现一个文件服务器 `file_server.js` ：

```
'use strict';

var
    fs = require('fs'),
    url = require('url'),
    path = require('path'),
    http = require('http');

// 从命令行参数获取root目录，默认是当前目录:
var root = path.resolve(process.argv[2] || '.');

console.log('Static root dir: ' + root);

// 创建服务器:
var server = http.createServer(function (request, response) {
    // 获得URL的path，类似 '/css/bootstrap.css':
    var pathname = url.parse(request.url).pathname;
    // 获得对应的本地文件路径，类似 '/srv/www/css/bootstrap.css':
    var filepath = path.join(root, pathname);
    // 获取文件状态:
    fs.stat(filepath, function (err, stats) {
        if (!err && stats.isFile()) {
```

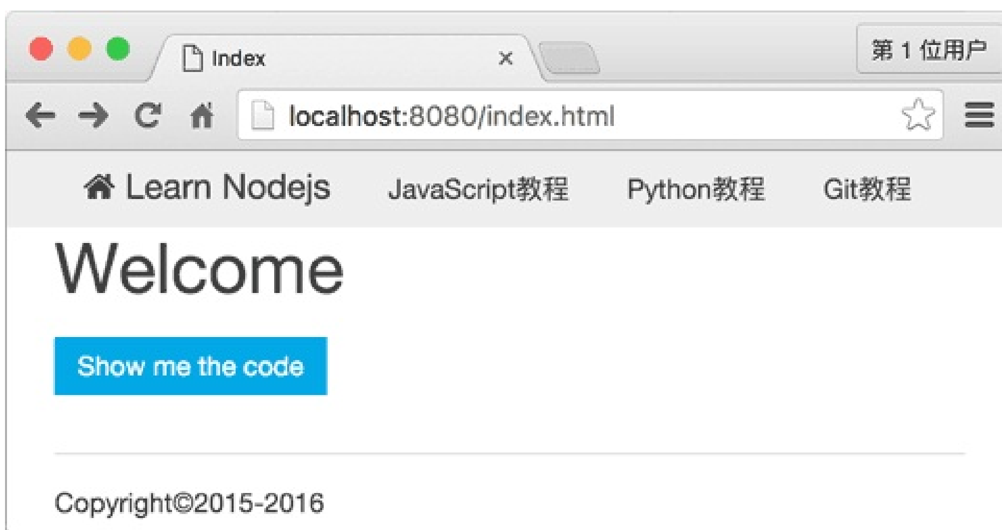
```
                // 没有出错并且文件存在:
                console.log('200 ' + request.url);
                // 发送200响应:
                response.writeHead(200);
                // 将文件流导向response:
                fs.createReadStream(filepath).pipe(response);
            } else {
                // 出错了或者文件不存在:
                console.log('404 ' + request.url);
                // 发送404响应:
                response.writeHead(404);
                response.end('404 Not Found');
            }
        });
});

server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');
```

没有必要手动读取文件内容。由于 `response` 对象本身是一个 `Writable Stream` ，直接用 `pipe()` 方法就实现了自动读取文件内容并输出到HTTP响应。

在命令行运行 `node file_server.js /path/to/dir` ，把 `/path/to/dir` 改成你本地的一个有效的目录，然后在浏览器中输入 `http://localhost:8080/index.html` ：



只要当前目录下存在文件 `index.html` ，服务器就可以把文件内容发送给浏览器。观察控制台输出：

```
200 /index.html
200 /css/uikit.min.css
200 /js/jquery.min.js
200 /fonts/fontawesome-webfont.woff2
```

第一个请求是浏览器请求 `index.html` 页面，后续请求是浏览器解析HTML后发送的其它资源请求。

练习

在浏览器输入 `http://localhost:8080/` 时，会返回404，原因是程序识别出HTTP请求的不是文件，而是目录。请修改 `file_server.js` ，如果遇到请求的路径是目录，则自动在目录下依次搜索 `index.html` 、 `default.html` ，如果找到了，就返回HTML文件的内容。

参考源码

http服务器代码（含静态网站）

buffer

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

Web开发

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

koa

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

mysql

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

swig

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

自动化工具

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

期末总结

即将推出，请耐心等待。等待不耐烦的，请关注微博@廖雪峰

Python 2.7教程

这是小白的Python新手教程。

Python是一种计算机程序设计语言。你可能已经听说过很多种流行的编程语言，比如非常难学的C语言，非常流行的Java语言，适合初学者的Basic语言，适合网页编程的JavaScript语言等等。

那Python是一种什么语言？

首先，我们普及一下编程语言的基础知识。用任何编程语言来开发程序，都是为了让计算机干活，比如下载一个MP3，编写一个文档等等，而计算机干活的CPU只认识机器指令，所以，尽管不同的编程语言差异极大，最后都得“翻译”成CPU可以执行的机器指令。而不同的编程语言，干同一个活，编写的代码量，差距也很大。

比如，完成同一个任务，C语言要写1000行代码，Java只需要写100行，而Python可能只要20行。

所以Python是一种相当高级的语言。

你也许会问，代码少还不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的Python程序设计也是非常难学的，所以，高级程序语言不等于简单。

但是，对于初学者和完成普通任务，Python语言是非常简单易用的。连Google都在大规模使用Python，你就不用担心学了会没用。

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，很多著名的网站包括YouTube就是Python写的；可以做网络游戏的后台，很多在线游戏的后台都是Python开发的。总之就是能干很多很多事啦。

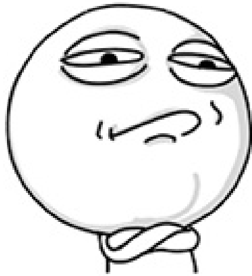
Python当然也有不能干的事情，比如写操作系统，这个只能用C语言写；写手机应用，只能用Objective-C（针对iPhone）和Java（针对Android）；写3D游戏，最好用C或C++。

如果你是小白用户，满足以下条件：

- 会使用电脑，但从来没写过程序；
- 还记得初中数学学的方程式和一点点代数知识；
- 想从编程小白变成专业的软件架构师；
- 每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

准备好了吗？



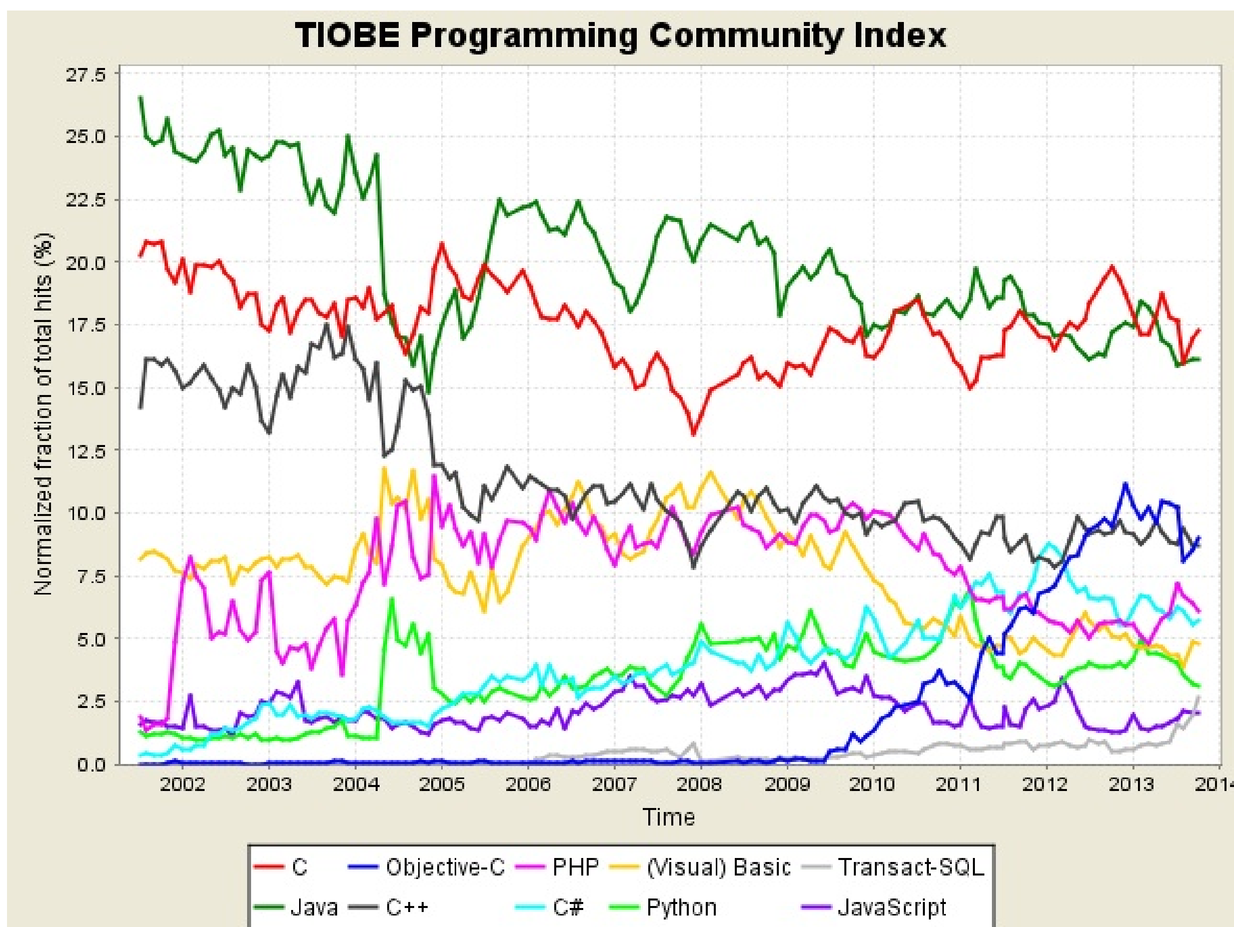
关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Visual Basic/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub，欢迎微博交流：@廖雪峰。

Python简介

Python是著名的“龟叔”Guido van Rossum在1989年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。

现在，全世界差不多有600多种编程语言，但流行的编程语言也就那么20来种。如果你听说过TIOBE排行榜，你就能知道编程语言的大致流行程度。这是最近10年最常用的10种编程语言的变化图：



总的来说，这几种编程语言各有千秋。C语言是可以用来编写操作系统的贴近硬件的语言，所以，C语言适合开发那些追求运行速度、充分发挥硬件性能的程序。而Python是用来编写应用程序的高级编程语言。

当你用一种语言开始作真正的软件开发时，你除了编写代码外，还需要很多基本的已经写好的现成的东西，来帮助你加快开发进度。比如说，要编写一个电子邮件客户端，如果先从最底层开始编写网络协议相关的代码，那估计一年半载也开发不出

来。高级编程语言通常都会提供一个比较完善的基础代码库，让你能直接调用，比如，针对电子邮件协议的SMTP库，针对桌面环境的GUI库，在这些已有的代码库的基础上开发，一个电子邮件客户端几天就能开发出来。

Python就为我们提供了非常完善的基础代码库，覆盖了网络、文件、GUI、数据库、文本等大量内容，被形象地称作“内置电池（batteries included）”。用Python开发，许多功能不必从零编写，直接使用现成的即可。

除了内置的库外，Python还有大量的第三方库，也就是别人开发的，供你直接使用的东西。当然，如果你开发的代码通过很好的封装，也可以作为第三方库给别人使用。

许多大型网站就是用Python开发的，例如YouTube、Instagram，还有国内的豆瓣。很多大公司，包括Google、Yahoo等，甚至NASA（美国航空航天局）都大量地使用Python。

龟叔给Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总的来说，Python的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那Python适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说Python的缺点。

任何编程语言都有缺点，Python也不例外。优点说过了，那Python有哪些缺点呢？

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

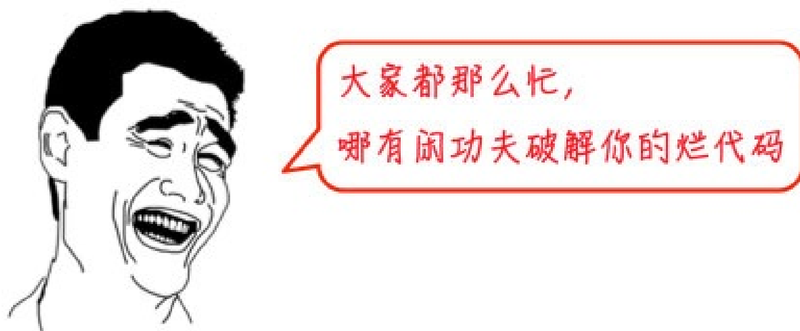
但是大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载MP3的网络应用程序，C程序的运行时间需要0.001秒，而Python程序的运行时间需要0.1秒，慢了100倍，但由于网络更慢，需要等待1秒，你想，用户能感觉到1.001秒和1.1秒的区别吗？这就好比F1赛车和普通的出租车在北京三环路上行驶的道理一样，虽然F1赛车理论时速高达400公里，但由于三环路堵车的时速只有20公里，因此，作为乘客，你感觉的时速永远是20公里。



第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

这个缺点仅限于你要编写的软件需要卖给别人挣钱的时候。好消息是目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站和移动应用卖服务的模式越来越多了，后一种模式不需要把源码给别人。

再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux一样的开源代码，我们千万不要高估自己写的代码真的有非常大的“商业价值”。那些大公司的代码不愿意开放的更重要的原因是代码写得太烂了，一旦开源，就没人敢用他们的产品了。



当然，Python还有其他若干小缺点，请自行忽略，就不一一列举了。

安装Python

因为Python是跨平台的，它可以运行在Windows、Mac和各种Linux/Unix系统上。在Windows上写Python程序，放到Linux上也是能够运行的。

要开始学习Python编程，首先就得把Python安装到你的电脑里。安装后，你会得到Python解释器（就是负责运行Python程序的），一个命令行交互环境，还有一个简单的集成开发环境。

2.x还是3.x

目前，Python有两个版本，一个是2.x版，一个是3.x版，这两个版本是不兼容的，因为现在Python正在朝着3.x版本进化，在进化过程中，大量的针对2.x版本的代码要修改后才能运行，所以，目前有许多第三方库还暂时无法在3.x上使用。

为了保证你的程序能用到大量的第三方库，我们的教程仍以2.x版本为基础，确切地说，是2.7版本。请确保你的电脑上安装的Python版本是2.7.x，这样，你才能无痛学习这个教程。

在Mac上安装Python

如果你正在使用Mac，系统是OS X 10.8或者最新的10.9 Mavericks，恭喜你，系统自带了Python 2.7。如果你的系统版本低于10.8，请自行备份系统并免费升级到最新的10.9，就可以获得Python 2.7。

查看系统版本的办法是点击左上角的苹果图标，选择“关于本机”：



在Linux上安装Python

如果你正在使用Linux，那我可以假定你有Linux系统管理经验，自行安装Python 2.7应该没有问题，否则，请换回Windows系统。

对于大量的目前仍在使用Windows的同学，如果短期内没有打算换Mac，就可以继续阅读以下内容。

在Windows上安装Python

首先，从Python的官方网站www.python.org下载最新的2.7.9版本，地址是这个：

http://www.python.org/ftp/python/2.7.9/python-2.7.9.msi

然后，运行下载的MSI安装包，在选择安装组件的一步时，勾上所有的组件：



特别要注意选上 `pip` 和 `Add python.exe to Path` ，然后一路点“Next”即可完成安装。

默认会安装到 `C:\Python27` 目录下，然后打开命令提示符窗口，敲入python后，会出现两种情况：

情况一：



看到上面的画面，就说明Python安装成功！

你看到提示符 >>> 就表示我们已经在Python交互式环境中了，可以输入任何Python代码，回车后会立刻得到执行结果。现在，输入 `exit()` 并回车，就可以退出Python交互式环境（直接关掉命令行窗口也可以！）。

情况二：得到一个错误：

```
‘python’不是内部或外部命令，也不是可运行的程序或批处理文件。
```

这是因为Windows会根据一个 `Path` 的环境变量设定的路径去查找 `python.exe` ，如果没找到，就会报错。如果在安装时漏掉了勾选 `Add python.exe to Path` ，那就要手动把 `python.exe` 所在的路径 `C:\Python27` 添加到Path中。

如果你不知道怎么修改环境变量，建议把Python安装程序重新运行一遍，记得勾上 `Add python.exe to Path` 。

小结

学会如何把Python安装到计算机中，并且熟练打开和退出Python交互式环境。

Python解释器

当我们编写Python代码时，我们得到的是一个包含Python代码的以 `.py` 为扩展名的文本文件。要运行代码，就需要Python解释器去执行 `.py` 文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

CPython

当我们从Python官方网站下载并安装好Python 2.7后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行 `python` 就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用 `>>>` 作为提示符，而IPython用 `In [序号]:` 作为提示符。

PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用JIT技术，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解PyPy和CPython的不同点。

Jython

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

IronPython

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

小结

Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 2.7版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

此外，教程还内嵌一个IPython的Web版本，用来在浏览器内练习执行一些Python代码。要注意两者功能一样，输入的代码一样，但是提示符有所不同。另外，不是所有代码都能在Web版本的IPython中执行，出于安全原因，很多操作（比如文件操作）是受限的，所以有些代码必须在本地环境执行代码。

第一个Python程序

现在，了解了如何启动和退出Python的交互式环境，我们就可以正式开始编写Python代码了。

在写代码之前，请千万不要用“复制”-“粘贴”把代码从页面粘贴到你自己的电脑上。写程序也讲究一个感觉，你需要一个字母一个字母地把代码自己敲进去，在敲代码的过程中，初学者经常会敲错代码，所以，你需要仔细地检查、对照，才能以最快的速度掌握如何写程序。

在交互式环境的提示符 `>>>` 下，直接输入代码，按回车，就可以立刻得到代码执行结果。现在，试试输入 `100+200` ，看看计算结果是不是300：

```
>>> 100+200
300
```

很简单吧，任何有效的数学计算都可以算出来。

如果要让Python打印出指定的文字，可以用 `print` 语句，然后把希望打印的文字用单引号或者双引号括起来，但不能混用单引号和双引号：

```
>>> print 'hello, world'
hello, world
```

这种用单引号或者双引号括起来的文本在程序中叫字符串，今后我们还会经常遇到。

最后，用 `exit()` 退出Python，我们的第一个Python程序完成！唯一的缺憾是没有保存下来，下次运行时还要再输入一遍代码。

小结

在Python交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。

使用文本编辑器

在Python的交互式命令行写程序，好处是一下就能得到结果，坏处是没法保存，下次还想运行的时候，还得再敲一遍。

所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

现在，我们就把上次的 `'hello, world'` 程序用文本编辑器写出来，保存下来。

那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是Sublime Text，免费使用，但是不付费会弹出提示框：



一个是Notepad++，免费使用，有中文界面：



请注意，用哪个都行，但是绝对不能用Word和Windows自带的记事本。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

```
print 'hello, world'
```

注意 `print` 前面不要有任何空格。然后，选择一个目录，例如 `C:\Workspace`，把文件保存为 `hello.py`，就可以打开命令行窗口，把当前目录切换到 `hello.py` 所在目录，就可以运行这个程序了：

```
C:\Workspace>python hello.py
hello, world
```

也可以保存为别的名字，比如 `abc.py`，但是必须要以 `.py` 结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有 `hello.py` 这个文件，运行 `python hello.py` 就会报错：

```
python hello.py
python: can't open file 'hello.py': [Errno 2] No such file or direc
```

报错的意思就是，无法打开 `hello.py` 这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

请注意区分命令行模式和Python交互模式：



看到类似 `C:\>` 是在Windows提供的命令行模式，看到 `>>>` 是在Python交互式环境下。

在命令行模式下，可以执行 `python` 进入Python交互式环境，也可以执行 `python hello.py` 运行一个 `.py` 文件，但是在Python交互式环境下，只能输入Python代码执行。

直接运行py文件

还有同学问，能不能像.exe文件那样直接运行 `.py` 文件呢？在Windows上是不行的，但是，在Mac和Linux上是可以的，方法是在 `.py` 文件的第一行加上：

```
#!/usr/bin/env python
```

然后，通过命令：

```
$ chmod a+x hello.py
```

就可以直接运行 `hello.py` 了，比如在Mac下运行：

```
Last login: Sun Dec 15 18:44:13 on ttys000
Mavericks:~ michael$ ./hello.py
hello, world
Mavericks:~ michael$
```

小结

用文本编辑器写Python程序，然后保存为后缀为 `.py` 的文件，就可以用Python直接运行这个程序了。

Python的交互模式和直接运行 `.py` 文件有什么区别呢？

直接输入 `python` 进入交互模式，相当于启动了Python解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行 `.py` 文件相当于启动了Python解释器，然后一次性把 `.py` 文件的源代码给执行了，你是没有机会输入源代码的。

用Python开发程序，完全可以一边在文本编辑器里写代码，一边开一个交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

输入和输出

输出

用 `print` 加上字符串，就可以向屏幕上输出指定的文字。比如输出 `'hello, world'`，用代码实现如下：

```
>>> print 'hello, world'
```

`print` 语句也可以跟上多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
>>> print 'The quick brown fox', 'jumps over', 'the lazy dog'
The quick brown fox jumps over the lazy dog
```

`print` 会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：



`print` 也可以打印整数，或者计算结果：

```
>>> print 300
300
>>> print 100 + 200
300
```

因此，我们可以把计算 `100 + 200` 的结果打印得更漂亮一点：

```
>>> print '100 + 200 =', 100 + 200
100 + 200 = 300
```

注意，对于 `100 + 200`，Python解释器自动计算出结果 `300`，但是，'100 + 200 ='是字符串而非数学公式，Python把它视为字符串，请自行解释上述打印结果。

输入

现在，你已经可以用 `print` 输出你想要的结果了。但是，如果要让用户从电脑输入一些字符怎么办？Python提供了一个 `raw_input` ，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

```
>>> name = raw_input()
Michael
```

当你输入 `name = raw_input()` 并按下回车后，Python交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

输入完成后，不会有任何提示，Python交互式命令行又回到 `>>>` 状态了。那我们刚才输入的内容到哪去了？答案是存放到 `name` 变量里了。可以直接输入 `name` 查看变量内容：

```
>>> name
'Michael'
```

什么是变量？请回忆初中数学所学的代数基础知识：

设正方形的边长为 `a` ，则正方形的面积为 `a x a` 。把边长 `a` 看做一个变量，我们就可以根据 `a` 的值计算正方形的面积，比如：

若a=2，则面积为a x a = 2 x 2 = 4；

若a=3.5，则面积为a x a = 3.5 x 3.5 = 12.25。

在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此， `name` 作为一个变量就是一个字符串。

要打印出 `name` 变量的内容，除了直接写 `name` 然后按回车外，还可以用 `print` 语句：

```
>>> print name
Michael
```

有了输入和输出，我们就可以把上次打印 `'hello, world'` 的程序改成有点意义的程序了：

```
name = raw_input()
print 'hello,', name
```

运行上面的程序，第一行代码会让用户输入任意字符作为自己的名字，然后存入 `name` 变量中；第二行代码会根据用户的名字向用户说 `hello` ，比如输入 `Michael` ：

```
C:\Workspace> python hello.py
Michael
hello, Michael
```

但是程序运行的时候，没有任何提示信息告诉用户："嘿，赶紧输入你的名字"，这样显得很不友好。幸好， `raw_input` 可以让你显示一个字符串来提示用户，于是我们把代码改成：

```
name = raw_input('please enter your name: ')
print 'hello,', name
```

再次运行这个程序，你会发现，程序一运行，会首先打印出 `please enter your name:` ，这样，用户就可以根据提示，输入名字后，得到 `hello, xxx` 的输出：

```
C:\Workspace> python hello.py
please enter your name: Michael
hello, Michael
```

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

小结

任何计算机程序都是为了执行一个特定的任务，有了输入，用户才能告诉计算机程序所需的信息，有了输出，程序运行后才能告诉用户任务的结果。

输入是Input，输出是Output，因此，我们把输入输出统称为Input/Output，或者简写为IO。

`raw_input` 和 `print` 是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

Python基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# print absolute value of an integer:
a = 100
if a >= 0:
    print a
else:
    print -a
```

以 `#` 开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号“:”结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

数据类型和变量

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

整数

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：`1`，`100`，`-8080`，`0`，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用`0x`前缀和0-9，a-f表示，例如：`0xff00`，`0xa5b4c3d2`，等等。

浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x10⁹和12.3x10⁸是相等的。浮点数可以用数学写法，如`1.23`，`3.14`，`-9.01`，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x10⁹就是`1.23e9`，或者`12.3e8`，0.000012可以写成`1.2e-5`，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

字符串

字符串是以"或""括起来的任意文本，比如`'abc'`，`"xyz"`等等。请注意，"或""本身只是一种表示方式，不是字符串的一部分，因此，字符串`'abc'`只有`a`，`b`，`c`这3个字符。如果`'`本身也是一个字符，那就可以用""括起来，比如`"I'm OK"`包含的字符是`I`，`'`，`m`，空格，`O`，`K`这6个字符。

如果字符串内部既包含`'`又包含`"`怎么办？可以用转义字符`\`来标识，比如：

```
'I\'m \"OK\"!'
```

表示的字符串内容是：

```
I'm "OK"!
```

转义字符 `\` 可以转义很多字符，比如 `\n` 表示换行， `\t` 表示制表符，字符 `\` 本身也要转义，所以 `\\` 表示的字符就是 `\` ，可以在Python的交互式命令行用print打印字符串看看：

```
>>> print 'I\'m ok.'
I'm ok.
>>> print 'I\'m learning\nPython.'
I'm learning
Python.
>>> print '\\\n\\'
\
\
```

如果字符串里面有很多字符都需要转义，就需要加很多 `\` ，为了简化，Python还允许用 `r''` 表示 `''` 内部的字符串默认不转义，可以自己试试：

```
>>> print '\\\t\\'
\       \
>>> print r'\\\t\\'
\\\t\\
```

如果字符串内部有很多换行，用 `\n` 写在一行里不好阅读，为了简化，Python允许用 `'''...'''` 的格式表示多行内容，可以自己试试：

```
>>> print '''line1
... line2
... line3'''
line1
line2
line3
```

上面是在交互式命令行内输入，如果写成程序，就是：

```
print '''line1
line2
line3'''
```

多行字符串 `'''...'''` 还可以在前面加上 `r` 使用，请自行测试。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有 `True` 、 `False` 两种值，要么是 `True` ，要么是 `False` ，在Python中，可以直接用 `True` 、 `False` 表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用 `and` 、 `or` 和 `not` 运算。

`and` 运算是与运算，只有所有都为 `True` ， `and` 运算结果才是 `True` ：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

`or` 运算是或运算，只要其中有一个为 `True` ， `or` 运算结果就是 `True` ：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

`not` 运算是非运算，它是一个单目运算符，把 `True` 变成 `False` ， `False` 变成 `True` ：

```
>>> not True
False
>>> not False
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:
    print 'adult'
else:
    print 'teenager'
```

空值

空值是Python里一个特殊的值，用 `None` 表示。 `None` 不能理解为 `0` ，因为 `0` 是有意义的，而 `None` 是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和 `_` 的组合，且不能用数字开头，比如：

```
a = 1
```

变量 `a` 是一个整数。

```
t_007 = 'T007'
```

变量 `t_007` 是一个字符串。

```
Answer = True
```

变量 `Answer` 是一个布尔值 `True` 。

在Python中，等号 `=` 是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

```
a = 123 # a是整数
print a
a = 'ABC' # a变为字符串
print a
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
x = 10
x = x + 2
```

如果从数学上理解 `x = x + 2` 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 `x + 2` ，得到结果 `12` ，再赋给变量 `x` 。由于 `x` 之前的值是 `10` ，重新赋值后， `x` 的值变成 `12` 。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python解释器干了两件事情：

1. 在内存中创建了一个 `'ABC'` 的字符串；
2. 在内存中创建了一个名为 `a` 的变量，并把它指向 `'ABC'` 。

也可以把一个变量 `a` 赋值给另一个变量 `b` ，这个操作实际上是把变量 `b` 指向变量 `a` 所指向的数据，例如下面的代码：

```
a = 'ABC'
b = a
a = 'XYZ'
print b
```

最后一行打印出变量 `b` 的内容到底是 `'ABC'` 呢还是 `'XYZ'` ？如果从数学意义上理解，就会错误地得出 `b` 和 `a` 相同，也应该是 `'XYZ'` ，但实际上 `b` 的值是 `'ABC'` ，让我们一行一行地执行代码，就可以看到到底发生了什么事：

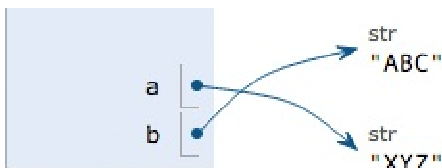
执行 `a = 'ABC'` ，解释器创建了字符串 `'ABC'` 和变量 `a` ，并把 `a` 指向 `'ABC'` ：



执行 `b = a` ，解释器创建了变量 `b` ，并把 `b` 指向 `a` 指向的字符串 `'ABC'` ：



执行 `a = 'XYZ'` ，解释器创建了字符串'XYZ'，并把 `a` 的指向改为 `'XYZ'` ，但 `b` 并没有更改：



所以，最后打印变量 `b` 的结果自然是 `'ABC'` 了。

常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

```
PI = 3.14159265359
```

但事实上 `PI` 仍然是一个变量，Python根本没有任何机制保证 `PI` 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 `PI` 的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的，可以试试：

```
>>> 10 / 3
3
```

你没有看错，整数除法永远是整数，即使除不尽。要做精确的除法，只需把其中一个整数换成浮点数做除法就可以：

```
>>> 10.0 / 3
3.3333333333333335
```

因为整数除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

```
>>> 10 % 3
1
```

无论整数做除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

小结

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

字符串和编码

字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是 `65535` ，4个字节可以表示的最大整数是 `4294967295` 。

由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 `ASCII` 编码，比如大写字母 `A` 的编码是 `65` ，小写字母 `z` 的编码是 `122` 。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了 `GB2312` 编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到 `Shift_JIS` 里，韩国把韩文编到 `Euc-kr` 里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。



因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母 `A` 用ASCII编码是十进制的 `65` ，二进制的 `01000001` ；

字符 `0` 用ASCII编码是十进制的 `48` ，二进制的 `00110000` ，注意字符 `'0'` 和整数 `0` 是不同的；

汉字 `中` 已经超出了ASCII编码的范围，用Unicode编码是十进制的 `20013` ，二进制的 `01001110 00101101` 。

你可以猜测，如果把ASCII编码的 `A` 用Unicode编码，只需要在前面补0就可以，因此， `A` 的Unicode编码是 `00000000 01000001` 。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的 `UTF-8` 编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

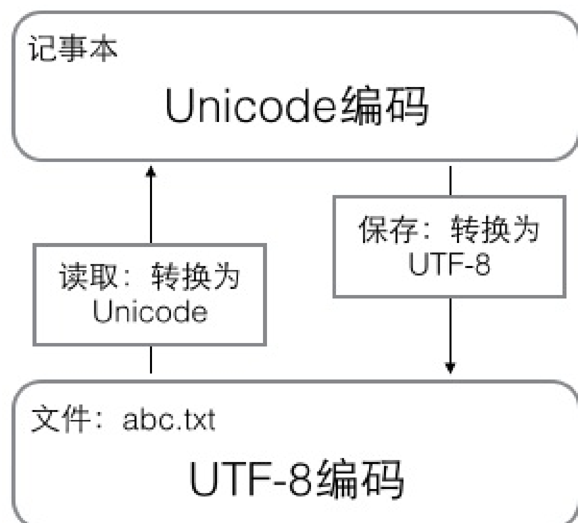
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

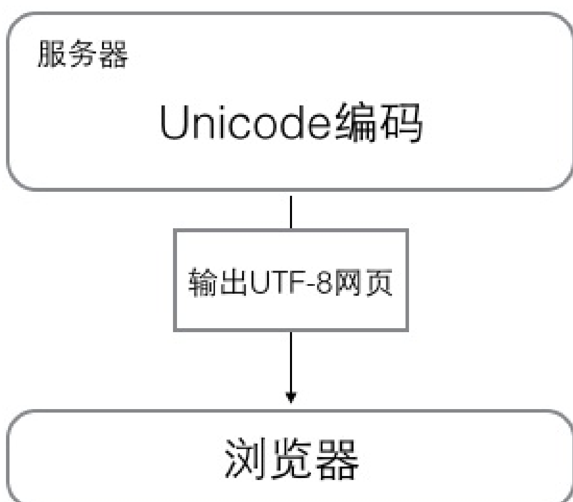
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看到很多网页的源码上会有类似 `<meta charset="UTF-8" />` 的信息，表示该网页正是用的UTF-8编码。

Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python对Unicode的支持。

因为Python的诞生比Unicode标准发布的时间还要早，所以最早的Python只支持ASCII编码，普通的字符串 `'ABC'` 在Python内部都是ASCII编码的。Python提供了ord()和chr()函数，可以把字母和对应的数字相互转换：

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Python在后来添加了对Unicode的支持，以Unicode表示的字符串用 `u'...'` 表示，比如：

```
>>> print u'中文'
中文
>>> u'中'
u'\u4e2d'
```

写 `u'中'` 和 `u'\u4e2d'` 是一样的， `\u` 后面是十六进制的Unicode码。因此， `u'A'` 和 `u'\u0041'` 也是一样的。

两种字符串如何相互转换？字符串 `'xxx'` 虽然是ASCII编码，但也可以看成是UTF-8编码，而 `u'xxx'` 则只能是Unicode编码。

把 `u'xxx'` 转换为UTF-8编码的 `'xxx'` 用 `encode('utf-8')` 方法：

```
>>> u'ABC'.encode('utf-8')
'ABC'
>>> u'中文'.encode('utf-8')
'\xe4\xb8\xad\xe6\x96\x87'
```

英文字符转换后表示的UTF-8的值和Unicode值相等（但占用的存储空间不同），而中文字符转换后1个Unicode字符将变为3个UTF-8字符，你看到的 `\xe4` 就是其中一个字节，因为它的值是 `228` ，没有对应的字母可以显示，所以以十六进制显示字节的数值。 `len()` 函数可以返回字符串的长度：

```
>>> len(u'ABC')
3
>>> len('ABC')
3
>>> len(u'中文')
2
>>> len('\xe4\xb8\xad\xe6\x96\x87')
6
```

反过来，把UTF-8编码表示的字符串 `'xxx'` 转换为Unicode字符串 `u'xxx'` 用 `decode('utf-8')` 方法：

```
>>> 'abc'.decode('utf-8')
u'abc'
>>> '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
u'\u4e2d\u6587'
>>> print '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
中文
```

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

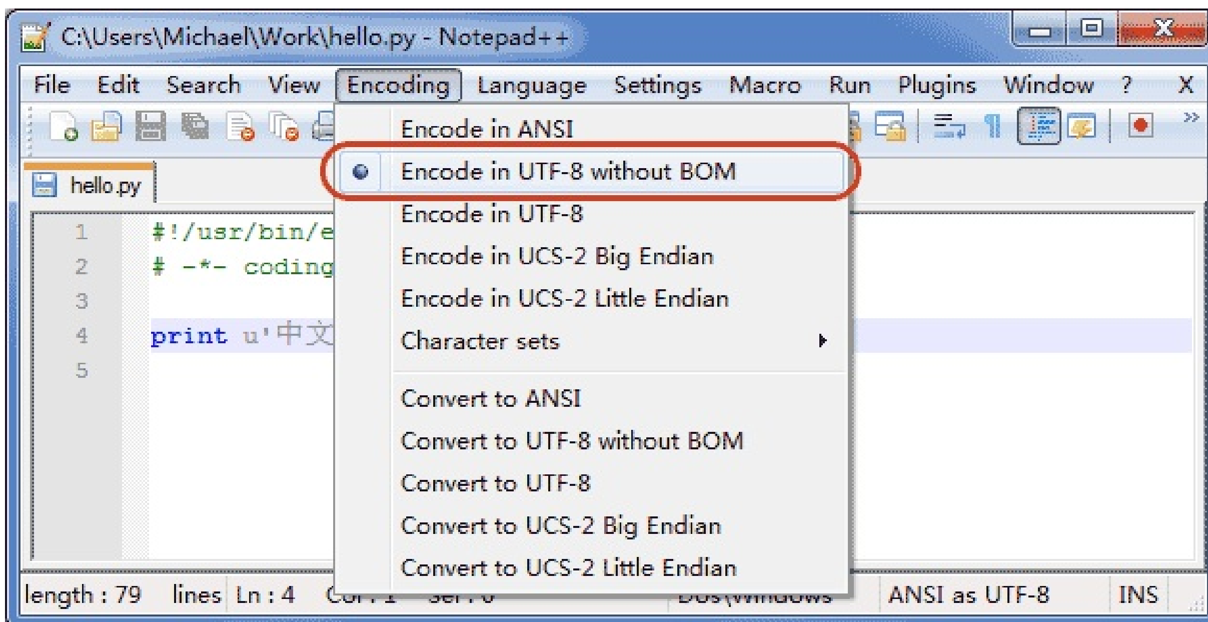
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

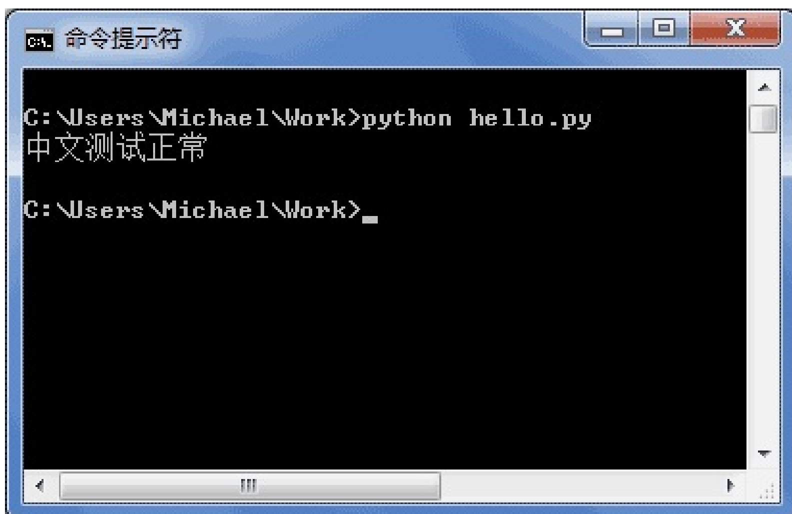
第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

如果你使用Notepad++进行编辑，除了要加上 `# -*- coding: utf-8 -*-` 外，中文字符串必须是Unicode字符串：

申明了UTF-8编码并不意味着你的 `.py` 文件就是UTF-8编码的，必须并且要确保Notepad++正在使用UTF-8 without BOM编码：

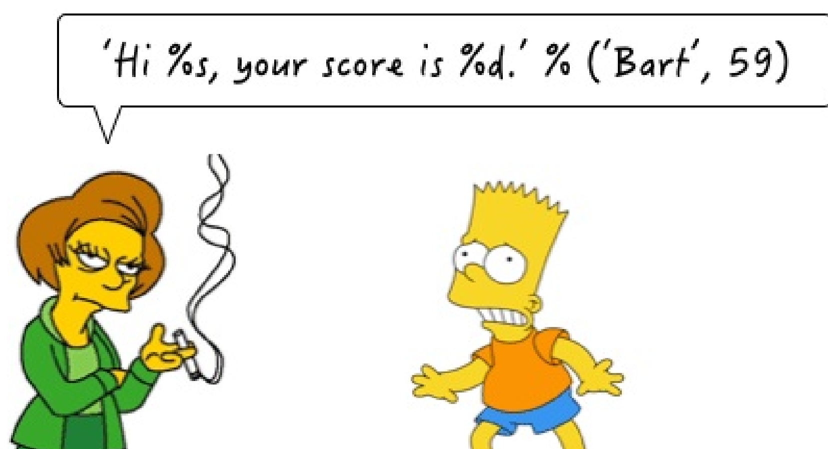


如果 `.py` 文件本身使用UTF-8编码，并且也申明了 `# -*- coding: utf-8 -*-` ，打开命令提示符测试就可以正常显示中文：



格式化

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似 `'亲爱的xxx你好！你xx月的话费是xx，余额是xx'` 之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。



在Python中，采用的格式化方式和C语言是一致的，用 `%` 实现，举例如下：

```
>>> 'Hello, %s' % 'world'
'Hello, world'
>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
'Hi, Michael, you have $1000000.'
```

你可能猜到了， `%` 运算符就是用来格式化字符串的。在字符串内部， `%s` 表示用字符串替换， `%d` 表示用整数替换，有几个 `%?` 占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个 `%?` ，括号可以省略。

常见的占位符有：

| %d | 整数 | | %f | 浮点数 | | %s | 字符串 | | %x | 十六进制整数 |

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

```
>>> '%2d-%02d' % (3, 1)
' 3-01'
>>> '%.2f' % 3.1415926
'3.14'
```

如果你不太确定应该用什么， `%s` 永远起作用，它会把任何数据类型转换为字符串：

```
>>> 'Age: %s. Gender: %s' % (25, True)
'Age: 25\. Gender: True'
```

对于Unicode字符串，用法完全一样，但最好确保替换的字符串也是Unicode字符串：

```
>>> u'Hi, %s' % u'Michael'
u'Hi, Michael'
```

有些时候，字符串里面的 `%` 是一个普通字符怎么办？这个时候就需要转义，用 `%%` 来表示一个 `%` ：

```
>>> 'growth rate: %d %%' % 7
'growth rate: 7 %'
```

小结

由于历史遗留问题，Python 2.x版本虽然支持Unicode，但在语法上需要 `'xxx'` 和 `u'xxx'` 两种字符串表示方式。

Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

```
>>> u'中文'.encode('gb2312')
'\xd6\xd0\xce\xc4'
```

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用Unicode和UTF-8这两种编码方式。

在Python 3.x版本中，把 `'xxx'` 和 `u'xxx'` 统一成Unicode编码，即写不写前缀 `u` 都是一样的，而以字节形式表示的字符串则必须加上 `b` 前缀： `b'xxx'` 。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

使用list和tuple

list

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

```
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
```

变量 `classmates` 就是一个list。用 `len()` 函数可以获得list元素的个数：

```
>>> len(classmates)
3
```

用索引来访问list中每一个位置的元素，记得索引是从 `0` 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是 `len(classmates) - 1` 。

如果要取最后一个元素，除了计算索引位置外，还可以用 `-1` 做索引，直接获取最后一个元素：

```
>>> classmates[-1]
'Tracy'
```

以此类推，可以获取倒数第2个、倒数第3个：

```
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为 `1` 的位置：

```
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除list末尾的元素，用 `pop()` 方法：

```
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

要删除指定位置的元素，用 `pop(i)` 方法，其中 `i` 是索引位置：

```
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

list里面的元素的数据类型也可以不同，比如：

```
>>> L = ['Apple', 123, True]
```

list元素也可以是另一个list，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
>>> len(s)
4
```

要注意 `s` 只有4个元素，其中 `s[2]` 又是一个list，如果拆开写就更容易理解了：

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
```

要拿到 `'php'` 可以写 `p[1]` 或者 `s[2][1]` ，因此 `s` 可以看成是一个二维数组，类似的还有三维、四维……数组，不过很少用到。

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

```
>>> L = []
>>> len(L)
0
```

tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用 `classmates[0]` ，`classmates[-1]` ，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

tuple的陷阱：当你定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来，比如：

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的tuple，可以写成 `()` ：

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有1个元素的tuple，如果你这么定义：

```
>>> t = (1)
>>> t
1
```

定义的不是tuple，是 `1` 这个数！这是因为括号 `()` 既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是 `1` 。

所以，只有1个元素的tuple定义时必须加一个逗号 `,` ，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

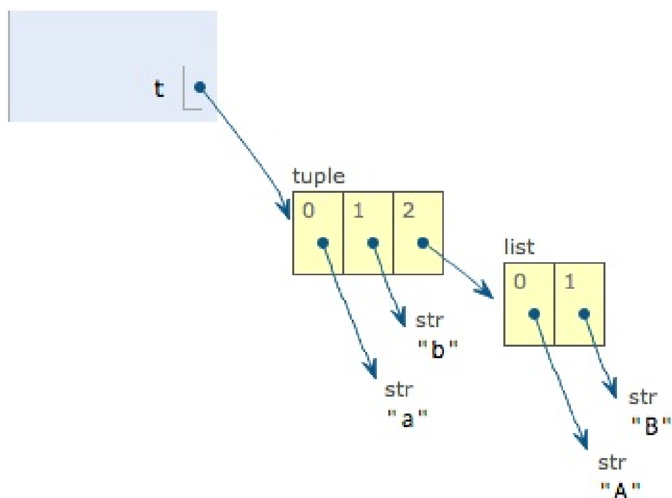
Python在显示只有1个元素的tuple时，也会加一个逗号 `,` ，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

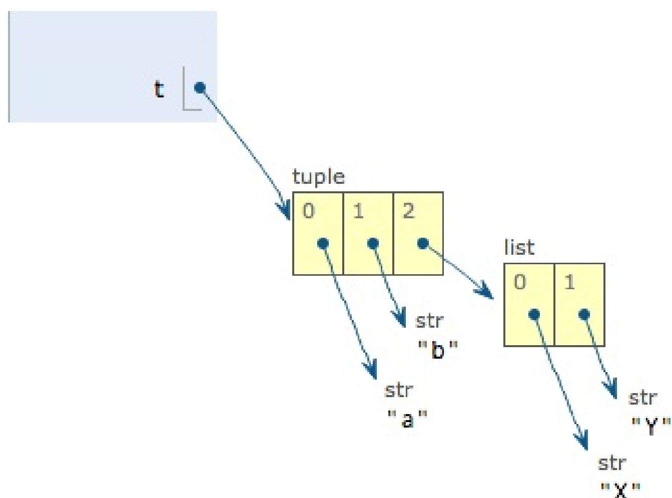
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

这个tuple定义的时候有3个元素，分别是 `'a'` ， `'b'` 和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素 `'A'` 和 `'B'` 修改为 `'X'` 和 `'Y'` 后，tuple变为：



表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向 `'a'` ，就不能改成指向 `'b'` ，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

小结

list和tuple是Python内置的有序集合，一个可变，一个不可变。根据需要来选择使用它们。

条件判断和循环

条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用 `if` 语句实现：

```
age = 20
if age >= 18:
    print 'your age is', age
    print 'adult'
```

根据Python的缩进规则，如果 `if` 语句判断是 `True` ，就把缩进的两行print语句执行了，否则，什么也不做。

也可以给 `if` 添加一个 `else` 语句，意思是，如果 `if` 判断是 `False` ，不要执行 `if` 的内容，去把 `else` 执行了：

```
age = 3
if age >= 18:
    print 'your age is', age
    print 'adult'
else:
    print 'your age is', age
    print 'teenager'
```

注意不要少写了冒号 `:` 。

当然上面的判断是很粗略的，完全可以用 `elif` 做更细致的判断：

```
age = 3
if age >= 18:
    print 'adult'
elif age >= 6:
    print 'teenager'
else:
    print 'kid'
```

`elif` 是 `else if` 的缩写，完全可以有多个 `elif` ，所以 `if` 语句的完整形式就是：

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

`if` 语句执行有个特点，它是从上往下判断，如果在某个判断上是 `True` ，把该判断对应的语句执行后，就忽略掉剩下的 `elif` 和 `else` ，所以，请测试并解释为什么下面的程序打印的是 `teenager` ：

```
age = 20
if age >= 6:
    print 'teenager'
elif age >= 18:
    print 'adult'
else:
    print 'kid'
```

`if` 判断条件还可以简写，比如写：

```
if x:
    print 'True'
```

只要 `x` 是非零数值、非空字符串、非空list等，就判断为 `True` ，否则为 `False` 。

循环

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print name
```

执行这段代码，会依次打印 `names` 的每一个元素：

```
Michael
Bob
Tracy
```

所以 `for x in ...` 循环就是把每个元素代入变量 `x` ，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个 `sum` 变量做累加：

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print sum
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range()函数，可以生成一个整数序列，比如range(5)生成的序列是从0开始小于5的整数：

```
>>> range(5)
[0, 1, 2, 3, 4]
```

range(101)就可以生成0-100的整数序列，计算如下：

```
sum = 0
for x in range(101):
    sum = sum + x
print sum
```

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print sum
```

在循环内部变量 `n` 不断自减，直到变为 `-1` 时，不再满足while条件，循环退出。

再议raw_input

最后看一个有问题的条件判断。很多同学会用 `raw_input()` 读取用户的输入，这样可以自己输入，程序运行得更有意思：

```
birth = raw_input('birth: ')
if birth < 2000:
    print '00前'
else:
    print '00后'
```

输入 `1982` ，结果却显示 `00后` ，这么简单的判断Python也能搞错？

当然不是Python的问题，在Python的交互式命令行下打印 `birth` 看看：

```
>>> birth
'1982'
>>> '1982' < 2000
False
>>> 1982 < 2000
True
```

原因找到了！原来从 `raw_input()` 读取的内容永远以字符串的形式返回，把字符串和整数比较就不会得到期待的结果，必须先用 `int()` 把字符串转换为我们想要的整型：

```
birth = int(raw_input('birth: '))
```

再次运行，就可以得到正确地结果。但是，如果输入 `abc` 呢？又会得到一个错误信息：

```
Traceback (most recent call last):
  ...
ValueError: invalid literal for int() with base 10: 'abc'
```

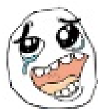
原来 `int()` 发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

小结

条件判断可以让计算机自己做选择，Python的if...elif...else很灵活。

if salary >= 10000:

print 

elif salary >=5000:

print 

else:

print 

循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用 `Ctrl+C` 退出程序，或者强制结束Python进程。

请试写一个死循环程序。

使用dict和set

dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字，无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如 `'Michael'` ，dict在内部就可以直接计算出 `Michael` 对应的存放成绩的“页码”，也就是 `95` 这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果key不存在，dict就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

要避免key不存在的错误，有两种办法，一是通过 `in` 判断key是否存在：

```
>>> 'Thomas' in d
False
```

二是通过dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value：

```
>>> d.get('Thomas')
>>> d.get('Thomas', -1)
-1
```

注意：返回None的时候Python的交互式命令行不显示结果。

要删除一个key，用 `pop(key)` 方法，对应的value也会从dict中删除：

```
>>> d.pop('Bob')
75
>>> d
{'Michael': 95, 'Tracy': 85}
```

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而增加；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是不可变对象。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

```
>>> key = [1, 2, 3]
>>> d[key] = 'a list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

```
>>> s = set([1, 2, 3])
>>> s
set([1, 2, 3])
```

注意，传入的参数 `[1, 2, 3]` 是一个list，而显示的 `set([1, 2, 3])` 只是告诉你这个set内部有1，2，3这3个元素，显示的[]不表示这是一个list。

重复元素在set中自动被过滤：

```
>>> s = set([1, 1, 2, 2, 3, 3])
>>> s
set([1, 2, 3])
```

通过 `add(key)` 方法可以添加元素到set中，可以重复添加，但不会有效果：

```
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
```

通过 `remove(key)` 方法可以删除元素：

```
>>> s.remove(4)
>>> s
set([1, 2, 3])
```

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
set([2, 3])
>>> s1 | s2
set([1, 2, 3, 4])
```

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

```
>>> a = ['c', 'b', 'a']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

而对于不可变对象，比如str，对str进行操作呢：

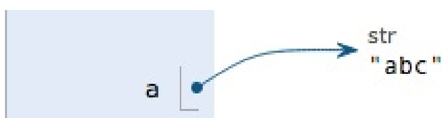
```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

虽然字符串有个 `replace()` 方法，也确实变出了 `'Abc'` ，但变量 `a` 最后仍是 `'abc'` ，应该怎么理解呢？

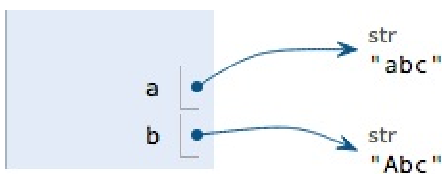
我们先把代码改成下面这样：

```
>>> a = 'abc'
>>> b = a.replace('a', 'A')
>>> b
'Abc'
>>> a
'abc'
```

要始终牢记的是， `a` 是变量，而 `'abc'` 才是字符串对象！有些时候，我们经常说，对象 `a` 的内容是 `'abc'` ，但其实是指， `a` 本身是一个变量，它指向的对象的内容才是 `'abc'` ：



当我们调用 `a.replace('a', 'A')` 时，实际上调用方法 `replace` 是作用在字符串对象 `'abc'` 上的，而这个方法虽然名字叫 `replace` ，但却没有改变字符串 `'abc'` 的内容。相反， `replace` 方法创建了一个新字符串 `'Abc'` 并返回，如果我们用变量 `b` 指向该新字符串，就容易理解了，变量 `a` 仍指向原有的字符串 `'abc'` ，但变量 `b` 却指向新字符串 `'Abc'` 了：



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把 `(1, 2, 3)` 和 `(1, [2, 3])` 放入dict或set中，并解释结果。

函数

我们知道圆的面积计算公式为：

S = πr²

当我们知道半径 `r` 的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把 `3.14` 改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x`，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：`1 + 2 + 3 + ... + 100`，写起来十分不方便，于是数学家发明了求和符号∑，可以把 `1 + 2 + 3 + ... + 100` 记作：

100

∑n

n=1

这种抽象记法非常强大，因为我们看到∑就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

∑(n²+1)

n=1

还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数 `abs` ，只有一个参数。可以直接从Python的官方网站查看文档：

http://docs.python.org/2/library/functions.html#abs

也可以在交互式命令行通过 `help(abs)` 查看 `abs` 函数的帮助信息。

调用 `abs` 函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

调用函数的时候，如果传入的参数数量不对，会报 `TypeError` 的错误，并且Python会明确地告诉你：abs()有且仅有1个参数，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 `TypeError` 的错误，并且给出错误信息：str是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而比较函数`cmp(x, y)`就需要两个参数，如果`x<y`，返回`-1`，如果`x==y`，返回`0`，如果`x>y`，返回`1`：

```
>>> cmp(1, 2)
-1
>>> cmp(2, 1)
1
>>> cmp(3, 3)
0
```

数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如`int()`函数可以把其他数据类型转换为整数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> unicode(100)
u'100'
>>> bool(1)
True
>>> bool('')
False
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量a指向abs函数
>>> a(-1) # 所以也可以通过a调用abs函数
1
```

小结

调用Python的函数，需要根据函数定义，传入正确的参数。如果函数调用出错，一定要学会看错误信息，所以英文很重要！

定义函数

在Python中，定义一个函数要使用 `def` 语句，依次写出函数名、括号、括号中的参数和冒号 `:`，然后，在缩进块中编写函数体，函数的返回值用 `return` 语句返回。

我们以自定义一个求绝对值的 `my_abs` 函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

请自行测试并调用 `my_abs` 看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后也会返回结果，只是结果为 `None` 。

`return None` 可以简写为 `return` 。

空函数

如果想定义一个什么事也不做的空函数，可以用 `pass` 语句：

```
def nop():
    pass
```

`pass` 语句什么都不做，那有什么用？实际上 `pass` 可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个 `pass` ，让代码能运行起来。

`pass` 还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了 `pass` ，代码运行就会有语法错误。

参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出 `TypeError` ：

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes exactly 1 argument (2 given)
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试 `my_abs` 和内置函数 `abs` 的差别：

```
>>> my_abs('A')
'A'
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时，内置函数 `abs` 会检查出参数错误，而我们定义的 `my_abs` 没有参数检查，所以，这个函数定义不够完善。

让我们修改一下 `my_abs` 的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数 `isinstance` 实现：

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续讲到。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

```
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

这样我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print x, y
151.961524227 70.0
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print r
(151.96152422706632, 70.0)
```

原来返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用 `return` 随时返回函数结果；

函数执行完毕也没有 `return` 语句时，自动 `return None` 。

函数可以同时返回多个值，但其实就是一个tuple。

函数的参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

默认参数

我们仍以具体的例子来说明如何定义函数的默认参数。先写一个计算x²的函数：

```
def power(x):
    return x * x
```

当我们调用 `power` 函数时，必须传入有且仅有的一个参数 `x` ：

```
>>> power(5)
25
>>> power(15)
225
```

现在，如果我们要计算x³怎么办？可以再定义一个 `power3` 函数，但是如果要计算x⁴、x⁵……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 `power(x)` 修改为 `power(x, n)` ，用来计算xⁿ，说干就干：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

对于这个修改后的 `power` 函数，可以计算任意n次方：

```
>>> power(5, 2)
25
>>> power(5, 3)
125
```

但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() takes exactly 2 arguments (1 given)
```

这个时候，默认参数就排上用场了。由于我们经常计算x²，所以，完全可以把第二个参数n的默认值设定为2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用 `power(5)` 时，相当于调用 `power(5, 2)` ：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于 `n > 2` 的其他情况，就必须明确地传入n，比如 `power(5, 3)` 。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入 `name` 和 `gender` 两个参数：

```
def enroll(name, gender):
    print 'name:', name
    print 'gender:', gender
```

这样，调用 `enroll()` 函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):
    print 'name:', name
    print 'gender:', gender
    print 'age:', age
    print 'city:', city
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
Student:
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)`，意思是，除了 `name`，`gender` 这两个参数外，最后1个参数应用在参数 `age` 上，`city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')`，意思是，`city` 参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个 `END` 再返回：

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 `add_end()` 时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是 `[]`，但是函数似乎每次都“记住了”上次添加了 `'END'` 后的list。

原因解释如下：

Python函数在定义的时候，默认参数 `L` 的值就被计算出来了，即 `[]`，因为默认参数 `L` 也是一个变量，它指向对象 `[]`，每次调用该函数，如果改变了 `L` 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的 `[]` 了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 `None` 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计str、None这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字a，b，c……，请计算a² + b² + c² + ……。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个list或tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义list或tuple参数相比，仅仅在参数前面加了一个 `*` 号。在函数内部，参数 `numbers` 接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个 `*` 号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

这种写法相当有用，而且很常见。

关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

```
def person(name, age, **kw):
    print 'name:', name, 'age:', age, 'other:', kw
```

函数 `person` 除了必选参数 `name` 和 `age` 外，还接受关键字参数 `kw` 。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在 `person` 函数里，我们保证能接收到 `name` 和 `age` 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=kw['city'], job=kw['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **kw)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这4种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。

比如定义一个函数，包含上述4种参数：

```
def func(a, b, c=0, *args, **kw):
    print 'a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> func(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> func(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> func(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> func(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

最神奇的是通过一个tuple和dict，你也可以调用该函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'x': 99}
>>> func(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'x': 99}
```

所以，对于任意函数，都可以通过类似 `func(*args, **kw)` 的形式调用它，无论它的参数是如何定义的。

小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

`*args` 是可变参数，args接收的是一个tuple；

`**kw` 是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：`func(1, 2, 3)`，又可以先组装list或tuple，再通过 `*args` 传入：`func(*(1, 2, 3))`；

关键字参数既可以直接传入：`func(a=1, b=2)`，又可以先组装dict，再通过 `**kw` 传入：`func(**{'a': 1, 'b': 2})`。

使用 `*args` 和 `**kw` 是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 `n! = 1 x 2 x 3 x ... x n`，用函数 `fact(n)` 表示，可以看出：

fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n

所以， `fact(n)` 可以表示为 `n x fact(n-1)`，只有n=1时需要特殊处理。

于是， `fact(n)` 用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217
```

如果我们计算 `fact(5)`，可以根据函数定义看到计算过程如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(1000)` ：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded
```

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)` 仅返回递归函数本身，`num - 1` 和 `num * product` 在函数调用前就会被计算，不影响函数调用。

`fact(5)` 对应的 `fact_iter(5, 1)` 的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

高级特性

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个`1, 3, 5, 7, ..., 99`的列表，可以通过循环实现：

```
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2
```

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，1行代码能实现的功能，决不写5行代码。

切片

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

取前3个元素，应该怎么做？

笨办法：

```
>>> [L[0], L[1], L[2]]
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

```
>>> r = []
>>> n = 3
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

`L[0:3]` 表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

```
>>> L[:3]
['Michael', 'Sarah', 'Tracy']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然Python支持 `L[-1]` 取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

记住倒数第一个元素的索引是 `-1` 。

切片操作十分有用。我们先创建一个0-99的数列：

```
>>> L = range(100)
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前10个数：

```
>>> L[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

后10个数：

```
>>> L[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

前11-20个数：

```
>>> L[10:20]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前10个数，每两个取一个：

```
>>> L[:10:2]
[0, 2, 4, 6, 8]
```

所有数，每5个取一个：

```
>>> L[::5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80,
```

甚至什么都不写，只写 `[:]` 就可以原样复制一个list：

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

```
>>> (0, 1, 2, 3, 4, 5)[:3]
(0, 1, 2)
```

字符串 `'xxx'` 或Unicode字符串 `u'xxx'` 也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

```
>>> 'ABCDEFG'[:3]
'ABC'
>>> 'ABCDEFG'[::2]
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数，其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

小结

有了切片操作，很多地方循环就不再需要了。Python的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

迭代

如果给定一个list或tuple，我们可以通过 `for` 循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过 `for ... in` 来完成的，而很多语言比如C或者Java，迭代list是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {
    n = list[i];
}
```

可以看出，Python的 `for` 循环抽象程度要高于Java的 `for` 循环，因为Python的 `for` 循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print key
...
a
c
b
```

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用 `for value in d.itervalues()` ，如果要同时迭代key和value，可以用 `for k, v in d.iteritems()` 。

由于字符串也是可迭代对象，因此，也可以作用于 `for` 循环：

```
>>> for ch in 'ABC':
...     print ch
...
A
B
C
```

所以，当我们使用`for`循环时，只要作用于一个可迭代对象，`for`循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str是否可迭代
True
>>> isinstance([1,2,3], Iterable) # list是否可迭代
True
>>> isinstance(123, Iterable) # 整数是否可迭代
False
```

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的`enumerate`函数可以把一个list变成索引-元素对，这样就可以在`for`循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):
...     print i, value
...
0 A
1 B
2 C
```

上面的`for`循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:
...     print x, y
...
1 1
2 4
3 9
```

小结

任何可迭代对象都可以作用于 `for` 循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用 `for` 循环。

列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` 可以用 `range(1, 11)` ：

```
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成 `[1x1, 2x2, 3x3, ..., 10x10]` 怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...    L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素 `x * x` 放到前面，后面跟 `for` 循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop',
```

`for` 循环其实可以同时使用两个甚至多个变量，比如 `dict` 的 `iteritems()` 可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.iteritems():
...     print k, '=', v
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.iteritems()]
['y=B', 'x=A', 'z=C']
```

最后把一个list中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

小结

运用列表生成式，可以快速生成list，可以通过一个list推导出另一个list，而代码却十分简洁。

思考：如果list中既包含字符串，又包含整数，由于非字符串类型没有 `lower()` 方法，所以列表生成式会报错：

```
>>> L = ['Hello', 'World', 18, 'Apple', None]
>>> [s.lower() for s in L]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

使用内建的 `isinstance` 函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
False
```

请修改列表生成式，通过添加 `if` 语句保证列表生成式能正确地执行。

生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器（Generator）。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的 `[]` 改成 `()` ，就创建了一个generator：

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x104feab40>
```

创建 `L` 和 `g` 的区别仅在于最外层的 `[]` 和 `()` ， `L` 是一个list，而 `g` 是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过generator的 `next()` 方法：

```
>>> g.next()
0
>>> g.next()
1
>>> g.next()
4
>>> g.next()
9
>>> g.next()
16
>>> g.next()
25
>>> g.next()
36
>>> g.next()
49
>>> g.next()
64
>>> g.next()
81
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

我们讲过，generator保存的是算法，每次调用 `next()` ，就计算出下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用 `next()` 方法实在是太变态了，正确的方法是使用 `for` 循环，因为generator也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print n
...
0
1
4
9
16
25
36
49
64
81
```

所以，我们创建了一个generator后，基本上永远不会调用 `next()` 方法，而是通过 `for` 循环来迭代它。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print b
        a, b = b, a + b
        n = n + 1
```

上面的函数可以输出斐波那契数列的前N个数：

```
>>> fib(6)
1
1
2
3
5
8
```

仔细观察，可以看出，`fib` 函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把 `fib` 函数变成generator，只需要把 `print b` 改为 `yield b` 就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
```

这就是定义generator的另一种方法。如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个generator：

```
>>> fib(6)
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句返回，再次执行时从上次返回的 `yield` 语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

```
>>> def odd():
...     print 'step 1'
...     yield 1
...     print 'step 2'
...     yield 3
...     print 'step 3'
...     yield 5
...
>>> o = odd()
>>> o.next()
step 1
1
>>> o.next()
step 2
3
>>> o.next()
step 3
5
>>> o.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，`odd`不是普通函数，而是generator，在执行过程中，遇到`yield`就中断，下次又继续执行。执行3次`yield`后，已经没有`yield`可以执行了，所以，第4次调用`next()`就报错。

回到`fib`的例子，我们在循环过程中不断调用`yield`，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用`next()`来调用它，而是直接使用`for`循环来迭代：

```
>>> for n in fib(6):
...     print n
...
1
1
2
3
5
8
```

小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在 `for` 循环的过程中不断计算出下一个元素，并在适当的条件结束 `for` 循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令， `for` 循环随之结束。

函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

高阶函数

高阶函数英文叫Higher-order function。什么是高阶函数？我们以实际代码为例子，一步一步深入概念。

变量可以指向函数

以Python内置的求绝对值的函数 `abs()` 为例，调用该函数用以下代码：

```
>>> abs(-10)
10
```

但是，如果只写 `abs` 呢？

```
>>> abs
<built-in function abs>
```

可见， `abs(-10)` 是函数调用，而 `abs` 是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量 `f` 现在已经指向了 `abs` 函数本身。

函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于 `abs()` 这个函数，完全可以把函数名 `abs` 看成变量，它指向一个可以计算绝对值的函数！

如果把 `abs` 指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 `abs` 指向 `10` 后，就无法通过 `abs(-10)` 调用该函数了！因为 `abs` 这个变量已经不指向求绝对值函数了！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复 `abs` 函数，请重启Python交互环境。

注：由于 `abs` 函数实际上是定义在 `__builtin__` 模块中的，所以要让修改 `abs` 变量的指向在其它模块也生效，要用 `__builtin__.abs = 10` 。

传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

当我们调用`add(-5, 6, abs)`时，参数`x`，`y`和`f`分别接收`-5`，`6`和`abs`，根据函数定义，我们可以推导计算过程为：

```
x ==> -5
y ==> 6
f ==> abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
```

用代码验证一下：

```
>>> add(-5, 6, abs)
11
```

编写高阶函数，就是让函数的参数能够接收别的函数。

小结

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

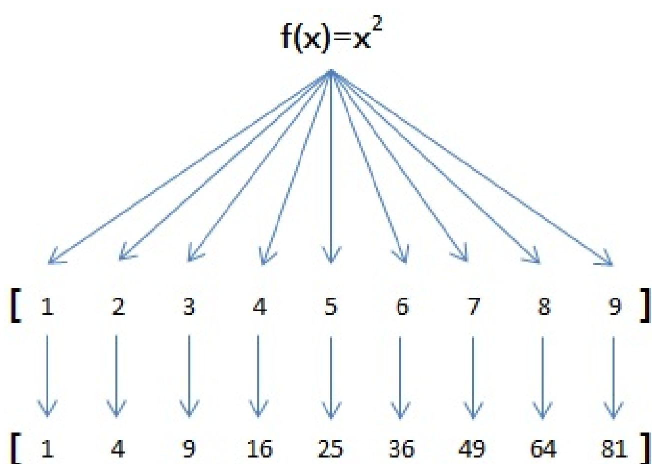
map/reduce

Python内建了 `map()` 和 `reduce()` 函数。

如果你读过Google的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。

我们先看map。 `map()` 函数接收两个参数，一个是函数，一个是序列， `map` 将传入的函数依次作用到序列的每个元素，并把结果作为新的list返回。

举例说明，比如我们有一个函数f(x)=x²，要把这个函数作用在一个list `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 上，就可以用 `map()` 实现如下：



现在，我们用Python代码实现：

```
>>> def f(x):
...     return x * x
...
>>> map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`map()` 传入的第一个参数是 `f` ，即函数对象本身。

你可能会想，不需要 `map()` 函数，写一个循环，也可以计算出结果：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print L
```

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以， `map()` 作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x²，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

```
>>> map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9])
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

只需要一行代码。

再看reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用reduce实现：

```
>>> def add(x, y):
...     return x + y
...
>>> reduce(add, [1, 3, 5, 7, 9])
25
```

当然求和运算可以直接用Python内建函数 `sum()` ，没必要动用reduce。

但是如果要把序列 `[1, 3, 5, 7, 9]` 变换成整数13579，reduce就可以派上用场：

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

这个例子本身没多大用处，但是，如果考虑到字符串 `str` 也是一个序列，对上面的例子稍加改动，配合 `map()` ，我们就可以写出把 `str` 转换为 `int` 的函数：

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
...     return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6
...
>>> reduce(fn, map(char2num, '13579'))
13579
```

整理成一个 `str2int` 的函数就是：

```
def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6
    return reduce(fn, map(char2num, s))
```

还可以用lambda函数进一步简化成：

```
def char2num(s):
    return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6,

def str2int(s):
    return reduce(lambda x,y: x*10+y, map(char2num, s))
```

也就是说，假设Python没有提供 `int()` 函数，你完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

lambda函数的用法在后面介绍。

练习

利用 `map()` 函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入： `['adam', 'LISA', 'barT']` ，输出： `['Adam', 'Lisa', 'Bart']` 。

Python提供的 `sum()` 函数可以接受一个list并求和，请编写一个 `prod()` 函数，可以接受一个list并利用 `reduce()` 求积。

filter

Python内建的 `filter()` 函数用于过滤序列。

和 `map()` 类似， `filter()` 也接收一个函数和一个序列。和 `map()` 不同的时， `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是 `True` 还是 `False` 决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1

filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])
# 结果: [1, 5, 9, 15]
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):
    return s and s.strip()

filter(not_empty, ['A', '', 'B', None, 'C', '  '])
# 结果: ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

练习

请尝试用 `filter()` 删除1~100的素数。

sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 `x` 和 `y` ，如果认为 `x < y` ，则返回 `-1` ，如果认为 `x == y` ，则返回 `0` ，如果认为 `x > y` ，则返回 `1` ，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python内置的 `sorted()` 函数就可以对list进行排序：

```
>>> sorted([36, 5, 12, 9, 21])
[5, 9, 12, 21, 36]
```

此外， `sorted()` 函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。比如，如果要倒序排序，我们就可以自定义一个 `reversed_cmp` 函数：

```
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0
```

传入自定义的比较函数 `reversed_cmp` ，就可以实现倒序排序：

```
>>> sorted([36, 5, 12, 9, 21], reversed_cmp)
[36, 21, 12, 9, 5]
```

我们再看一个字符串排序的例子：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])
['Credit', 'Zoo', 'about', 'bob']
```

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于 `'Z' < 'a'` ，结果，大写字母 `Z` 会排在小写字母 `a` 的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

```
def cmp_ignore_case(s1, s2):
    u1 = s1.upper()
    u2 = s2.upper()
    if u1 < u2:
        return -1
    if u1 > u2:
        return 1
    return 0
```

忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给 `sorted` 传入上述比较函数，即可实现忽略大小写的排序：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], cmp_ignore_case)
['about', 'bob', 'Credit', 'Zoo']
```

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

返回函数

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function sum at 0x10452f668>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum` ，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `args` ，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用`f1()`，`f2()`和`f3()`结果应该是`1`，`4`，`9`，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是`9`！原因就在于返回的函数引用了变量`i`，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量`i`已经变成了3，因此最终结果为`9`。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
>>> def count():
...     fs = []
...     for i in range(1, 4):
...         def f(j):
...             def g():
...                 return j*j
...             return g
...         fs.append(f(i))
...     return fs
...
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

缺点是代码较长，可利用lambda函数缩短代码。

匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以 `map()` 函数为例，计算f(x)=x²时，除了定义一个 `f(x)` 的函数外，还可以直接传入匿名函数：

```
>>> map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 `lambda x: x * x` 实际上就是：

```
def f(x):
    return x * x
```

关键字 `lambda` 表示匿名函数，冒号前面的 `x` 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写 `return` ，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x10453d7d0>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda: x * x + y * y
```

小结

Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print '2013-12-25'
...
>>> f = now
>>> f()
2013-12-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print 'call %s():' % func.__name__
        return func(*args, **kw)
    return wrapper
```

观察上面的 `log` ，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

```
@log
def now():
    print '2013-12-25'
```

调用 `now()` 函数，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志：

```
>>> now()
call now():
2013-12-25
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个decorator，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)` ，因此， `wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个3层嵌套的decorator用法如下：

```
@log('execute')
def now():
    print '2013-12-25'
```

执行结果如下：

```
>>> now()
execute now():
2013-12-25
```

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')` ，返回的是 `decorator` 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 `wrapper` 函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你去看经过decorator装饰之后的函数，它们的 `__name__` 已经从原来的 `'now'` 变成了 `'wrapper'` ：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 `wrapper()` 函数名字就是 `'wrapper'` ，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python内置的 `functools.wraps` 就是干这个事的，所以，一个完整的decorator的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print 'call %s():' % func.__name__
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的decorator：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出 `'begin call'` 和 `'end call'` 的日志。

再思考一下能否写出一个 `@log` 的decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

偏函数

Python的 `functools` 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

`int()` 函数可以把字符串转换为整数，当仅传入字符串时， `int()` 函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 `int()` 函数还提供额外的 `base` 参数，默认值为 `10` 。如果传入 `base` 参数，就可以做N进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()` ，可以直接使用下面的代码创建一个新的函数 `int2` ：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 `2` ，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
1000000
```

最后，创建偏函数时，实际上可以接收函数对象、 `*args` 和 `**kw` 这3个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了int()函数的关键字参数 `base` ，也就是：

```
int2('10010')
```

相当于：

```
kw = { base: 2 }
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 `10` 作为 `*args` 的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)
max(*args)
```

结果为 `10` 。

小结

当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

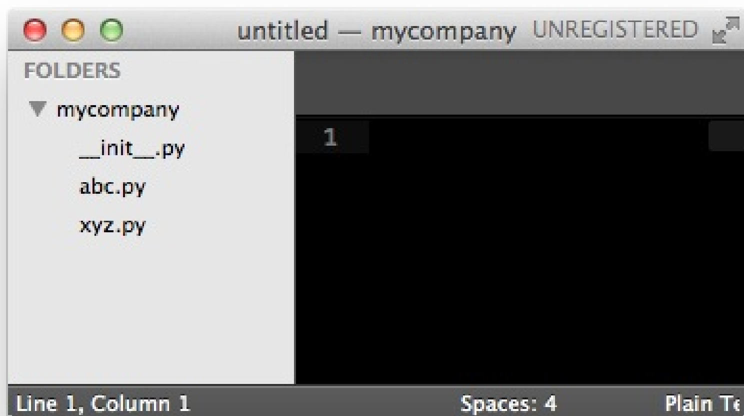
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点这里查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个 `abc.py` 的文件就是一个名字叫 `abc` 的模块，一个 `xyz.py` 的文件就是一个名字叫 `xyz` 的模块。

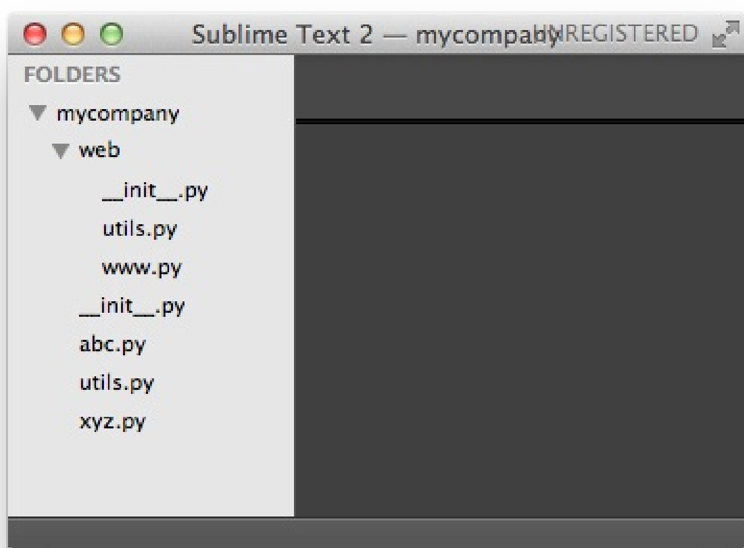
现在，假设我们的 `abc` 和 `xyz` 这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 `mycompany` ，按照如下目录存放：



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在， `abc.py` 模块的名字就变成了 `mycompany.abc` ，类似的， `xyz.py` 的模块名变成了 `mycompany.xyz` 。

请注意，每一个包目录下面都会有一个 `__init__.py` 的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。 `__init__.py` 可以是空文件，也可以有Python代码，因为 `__init__.py` 本身就是一个模块，而它的模块名就是 `mycompany` 。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



文件`www.py`的模块名就是`mycompany.web.www`，两个文件`utils.py`的模块名分别是`mycompany.utils`和`mycompany.web.utils`。

`mycompany.web`也是一个模块，请指出该模块对应的.py文件。

使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的 `sys` 模块为例，编写一个 `hello` 的模块：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
        print 'Hello, world!'
    elif len(args)==2:
        print 'Hello, %s!' % args[1]
    else:
        print 'Too many arguments!'

if __name__=='__main__':
    test()
```

第1行和第2行是标准注释，第1行注释可以让这个 `hello.py` 文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用 `__author__` 变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用 `sys` 模块的第一步，就是导入该模块：

```
import sys
```

导入 `sys` 模块后，我们就有了变量 `sys` 指向该模块，利用 `sys` 这个变量，就可以访问 `sys` 模块的所有功能。

`sys` 模块有一个 `argv` 变量，用list存储了命令行的所有参数。 `argv` 至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行 `python hello.py` 获得的 `sys.argv` 就是 `['hello.py']` ；

运行 `python hello.py Michael` 获得的 `sys.argv` 就是 `['hello.py', 'Michael]` 。

最后，注意到这两行代码：

```
if __name__=='__main__':
    test()
```

当我们在命令行运行 `hello` 模块文件时，Python解释器把一个特殊变量 `__name__` 置为 `__main__` ，而如果在其他地方导入该 `hello` 模块时， `if` 判断将失败，因此，这种 `if` 测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行 `hello.py` 看看效果：

```
$ python hello.py
Hello, world!
$ python hello.py Michael
Hello, Michael!
```

如果启动Python交互环境，再导入 `hello` 模块：

```
$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more informat:
>>> import hello
>>>
```

导入时，没有打印 `Hello, word!` ，因为没有执行 `test()` 函数。

调用 `hello.test()` 时，才能打印出 `Hello, word!` ：

```
>>> hello.test()
Hello, world!
```

别名

导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供 `StringIO` 和 `cStringIO` 两个库，这两个库的接口和功能是一样的，但是 `cStringIO` 是C写的，速度更快，所以，你会经常看到这样的写法：

```
try:
    import cStringIO as StringIO
except ImportError: # 导入失败会捕获到ImportError
    import StringIO
```

这样就可以优先导入 `cStringIO` 。如果有些平台不提供 `cStringIO` ，还可以降级使用 `StringIO` 。导入 `cStringIO` 时，用 `import ... as ...` 指定了别名 `StringIO` ，因此，后续代码引用 `StringIO` 即可正常工作。

还有类似 `simplejson` 这样的库，在Python 2.6之前是独立的第三方库，从2.6开始内置，所以，会有这样的写法：

```
try:
    import json # python >= 2.6
except ImportError:
    import simplejson as json # python <= 2.5
```

由于Python是动态语言，函数签名一致接口就一样，因此，无论导入哪个模块后续代码都能正常工作。

作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过 `_` 前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如： `abc` ， `x123` ， `PI` 等；

类似 `__xxx__` 这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的 `__author__` ， `__name__` 就是特殊变量， `hello` 模块定义的文档注释也可以用特殊变量 `__doc__` 访问，我们自己的变量一般不要用这种变量名；

类似 `_xxx` 和 `__xxx` 这样的函数或变量就是非公开的（private），不应该被直接引用，比如 `_abc` ， `__abc` 等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

```
def _private_1(name):
    return 'Hello, %s' % name

def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

我们在模块里公开 `greeting()` 函数，而把内部逻辑用private函数隐藏起来了，这样，调用 `greeting()` 函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

安装第三方模块

在Python中，安装第三方模块，是通过setuptools这个工具完成的。Python有两个封装了setuptools的包管理工具： `easy_install` 和 `pip` 。目前官方推荐使用 `pip` 。

如果你正在使用Mac或Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用Windows，请参考安装Python一节的内容，确保安装时勾选了 `pip` 和 `Add python.exe to Path` 。

在命令提示符窗口下尝试运行 `pip` ，如果Windows提示未找到命令，可以重新运行安装程序添加 `pip` 。

现在，让我们来安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。一般来说，第三方库都会在Python官方的pypi.python.org网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Python Imaging Library的名称叫PIL，因此，安装Python Imaging Library的命令就是：

```
pip install PIL
```

耐心等待下载并安装后，就可以使用PIL了。

有了PIL，处理图片易如反掌。随便找个图片生成缩略图：

```
>>> import Image
>>> im = Image.open('test.png')
>>> print im.format, im.size, im.mode
PNG (400, 300) RGB
>>> im.thumbnail((200, 100))
>>> im.save('thumb.jpg', 'JPEG')
```

其他常用的第三方库还有MySQL的驱动： `MySQL-python` ，用于科学计算的NumPy库： `numpy` ，用于生成文本的模板工具 `Jinja2` ，等等。

模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在 `sys` 模块的 `path` 变量中：

```
>>> import sys
>>> sys.path
['', '/Library/Python/2.7/site-packages/pycrypto-2.6.1-py2.7-macosx
```

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改 `sys.path` ，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量 `PYTHONPATH` ，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

使用__future__

Python的每个新版本都会增加一些新的功能，或者对原来的功能作一些改动。有些改动是不兼容旧版本的，也就是在当前版本运行正常的代码，到下一个版本运行就可能不正常了。

从Python 2.7到Python 3.x就有不兼容的一些改动，比如2.x里的字符串用 `'xxx'` 表示str，Unicode字符串用 `u'xxx'` 表示unicode，而在3.x中，所有字符串都被视为unicode，因此，写 `u'xxx'` 和 `'xxx'` 是完全一致的，而在2.x中以 `'xxx'` 表示的str就必须写成 `b'xxx'` ，以此表示“二进制字符串”。

要直接把代码升级到3.x是比较冒进的，因为有大量的改动需要测试。相反，可以在2.7版本中先在一部分代码中测试一些3.x的特性，如果没有问题，再移植到3.x不迟。

Python提供了 `__future__` 模块，把下一个新版本的特性导入到当前版本，于是我们就可以在当前版本中测试一些新版本的特性。举例说明如下：

为了适应Python 3.x的新的字符串的表示方法，在2.7版本的代码中，可以通过 `unicode_literals` 来使用Python 3.x的新的语法：

```
# still running on Python 2.7

from __future__ import unicode_literals

print '\'xxx\' is unicode?', isinstance('xxx', unicode)
print 'u\'xxx\' is unicode?', isinstance(u'xxx', unicode)
print '\'xxx\' is str?', isinstance('xxx', str)
print 'b\'xxx\' is str?', isinstance(b'xxx', str)
```

注意到上面的代码仍然在Python 2.7下运行，但结果显示去掉前缀 `u` 的 `'a string'` 仍是一个unicode，而加上前缀 `b` 的 `b'a string'` 才变成了str：

```
$ python task.py
'xxx' is unicode? True
u'xxx' is unicode? True
'xxx' is str? False
b'xxx' is str? True
```

类似的情况还有除法运算。在Python 2.x中，对于除法有两种情况，如果是整数相除，结果仍是整数，余数会被扔掉，这种除法叫“地板除”：

```
>>> 10 / 3
3
```

要做精确除法，必须把其中一个数变成浮点数：

```
>>> 10.0 / 3
3.3333333333333335
```

而在Python 3.x中，所有的除法都是精确除法，地板除用 `//` 表示：

```
$ python3
Python 3.3.2 (default, Jan 22 2014, 09:54:40)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more informati
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
```

如果你想在Python 2.7的代码中直接使用Python 3.x的除法，可以通过 `__future__` 模块的 `division` 实现：

```
from __future__ import division

print '10 / 3 =', 10 / 3
print '10.0 / 3 =', 10.0 / 3
print '10 // 3 =', 10 // 3
```

结果如下：

```
10 / 3 = 3.33333333333
10.0 / 3 = 3.33333333333
10 // 3 = 3
```

小结

由于Python是由社区推动的开源并且免费的开发语言，不受商业公司控制，因此，Python的改进往往比较激进，不兼容的情况时有发生。Python为了确保你能顺利过渡到新版本，特别提供了 `__future__` 模块，让你在旧的版本中试验新版本的一些特性。

面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
def print_score(std):
    print '%s: %s' % (std['name'], std['score'])
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 `Student` 这种数据类型应该被视为一个对象，这个对象拥有 `name` 和 `score` 这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 `print_score` 消息，让对象自己把自己的数据打印出来。

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print '%s: %s' % (self.name, self.score)
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student：

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student类为例，在Python中，定义类是通过 `class` 关键字：

```
class Student(object):
    pass
```

`class` 后面紧接着是类名，即 `Student` ，类名通常是大写开头的单词，紧接着是 `(object)` ，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用 `object` 类，这是所有类最终都会继承的类。

定义好了 `Student` 类，就可以根据 `Student` 类创建出 `Student` 的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量 `bart` 指向的就是一个Student的object，后面的 `0x10a67a590` 是内存地址，每个object的地址都不一样，而 `Student` 本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例 `bart` 绑定一个 `name` 属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 `__init__` 方法，在创建实例的时候，就把 `name` ， `score` 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意到 `__init__` 方法的第一个参数永远是 `self` ，表示创建的实例本身，因此，在 `__init__` 方法内部，就可以把各种属性绑定到 `self` ，因为 `self` 就指向创建的实例本身。

有了 `__init__` 方法，在创建实例的时候，就不能传入空的参数了，必须传入与 `__init__` 方法匹配的参数，但 `self` 不需要传，Python解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 `self` ，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数和关键字参数。

数据封装

面向对象编程的一个重要特点就是数据封装。在上面的 `Student` 类中，每个实例就拥有各自的 `name` 和 `score` 这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

```
>>> def print_score(std):
...     print '%s: %s' % (std.name, std.score)
...
>>> print_score(bart)
Bart Simpson: 59
```

但是，既然 `Student` 实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在 `Student` 类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和 `Student` 类本身是关联起来的，我们称之为类的方法：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print '%s: %s' % (self.name, self.score)
```

要定义一个方法，除了第一个参数是 `self` 外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了 `self` 不用传递，其他参数正常传入：

```
>>> bart.print_score()
Bart Simpson: 59
```

这样一来，我们从外部看 `Student` 类，就只需要知道，创建实例需要给出 `name` 和 `score` ，而如何打印，都是在 `Student` 类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给 `Student` 类增加新的方法，比如 `get_grade` ：

```
class Student(object):
    ...

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同样的，`get_grade`方法可以直接在实例变量上调用，不需要知道内部实现细节：

```
>>> bart.get_grade()
'C'
```

小结

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

```
>>> bart = Student('Bart Simpson', 59)
>>> lisa = Student('Lisa Simpson', 87)
>>> bart.age = 8
>>> bart.age
8
>>> lisa.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的 `name` 、 `score` 属性：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.score
98
>>> bart.score = 59
>>> bart.score
59
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线 `__` ，在Python中，实例的变量名如果以 `__` 开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print '%s: %s' % (self.__name, self.__score)
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问 `实例变量.__name` 和 `实例变量.__score` 了：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute '__name'
```

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加 `get_name` 和 `get_score` 这样的方法：

```
class Student(object):
    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score
```

如果又要允许外部代码修改score怎么办？可以给Student类增加 `set_score` 方法：

```
class Student(object):
    ...

    def set_score(self, score):
        self.__score = score
```

你也许会问，原先那种直接通过 `bart.score = 59` 也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

```
class Student(object):
    ...

    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

需要注意的是，在Python中，变量名类似 `__xxx__` 的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用 `__name__` 、 `__score__` 这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如 `_name` ，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问 `__name` 是因为Python解释器对外把 `__name` 变量改成了 `_Student__name` ，所以，仍然可以通过 `_Student__name` 来访问 `__name` 变量：

```
>>> bart._Student__name
'Bart Simpson'
```

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把 `__name` 改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为 `Animal` 的class，有一个 `run()` 方法可以直接打印：

```
class Animal(object):
    def run(self):
        print 'Animal is running...'
```

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。Cat和Dog类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于Animial实现了 `run()` 方法，因此，Dog和Cat作为它的子类，什么事也没干，就自动拥有了 `run()` 方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如Dog类：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'
    def eat(self):
        print 'Eating meat...'
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是Dog还是Cat，它们 `run()` 的时候，显示的都是 `Animal is running...` ，符合逻辑的做法是分别显示 `Dog is running...` 和 `Cat is running...` ，因此，对Dog和Cat类改进如下：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'

class Cat(Animal):
    def run(self):
        print 'Cat is running...'
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类和父类都存在相同的 `run()` 方法时，我们说，子类的 `run()` 覆盖了父类的 `run()` ，在代码运行的时候，总是会调用子类的 `run()` 。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 `isinstance()` 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来a、b、c确实对应着list、Animal、Dog这3种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来c不仅仅是Dog，c还是Animal！

不过仔细想想，这是有道理的，因为Dog是从Animal继承下来的，当我们创建了一个Dog的实例 `c` 时，我们认为 `c` 的数据类型是Dog没错，但 `c` 同时也是Animal也没错，Dog本来就是Animal的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

Dog可以看成Animal，但Animal不可以看成Dog。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Animal类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入Animal的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入Dog的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入Cat的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

```
class Tortoise(Animal):
    def run(self):
        print 'Tortoise is running slowly...'
```

当我们调用run_twice()时，传入Tortoise的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个Animal的子类，不必对run_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

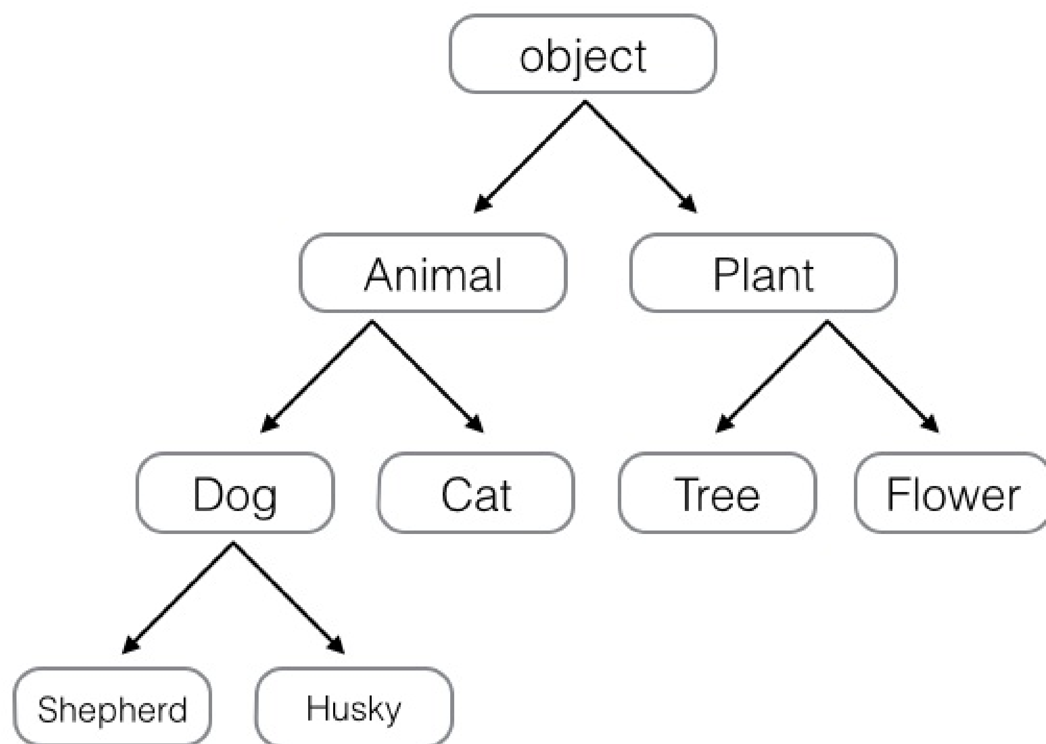
多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有 `run()` 方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用 `run()` 方法，而具体调用的 `run()` 方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保 `run()` 方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的 `run_twice()` 等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写；

有了继承，才能有多态。在调用类实例方法的时候，尽量把变量视作父类类型，这样，所有子类类型都可以正常被接收；

旧的方式定义Python类允许不从object类继承，但这种编程方式已经严重不推荐使用。任何时候，如果没有合适的类可以继承，就继承自object类。

获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用type()

首先，我们来判断对象类型，使用 `type()` 函数：

基本类型都可以用 `type()` 判断：

```
>>> type(123)
<type 'int'>
>>> type('str')
<type 'str'>
>>> type(None)
<type 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 `type()` 判断：

```
>>> type(abs)
<type 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是 `type()` 函数返回的是什么类型呢？它返回type类型。如果我们要在 `if` 语句中判断，就需要比较两个变量的type类型是否相同：

```
>>> type(123)==type(456)
True
>>> type('abc')==type('123')
True
>>> type('abc')==type(123)
False
```

但是这种写法太麻烦，Python把每种type类型都定义好了常量，放在 `types` 模块里，使用之前，需要先导入：

```
>>> import types
>>> type('abc')==types.StringType
True
>>> type(u'abc')==types.UnicodeType
True
>>> type([])==types.ListType
True
>>> type(str)==types.TypeType
True
```

最后注意到有一种类型就叫 `TypeType` ，所有类型本身的类型就是 `TypeType` ，比如：

```
>>> type(int)==type(str)==types.TypeType
True
```

使用isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用 `isinstance()` 函数。

我们回顾上次的例子，如果继承关系是：

```
object -> Animal -> Dog -> Husky
```

那么， `isinstance()` 就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为 `h` 变量指向的就是Husky对象。

再判断：

```
>>> isinstance(h, Dog)
True
```

`h` 虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以， `h` 也还是Dog类型。换句话说， `isinstance()` 判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信， `h` 还是Animal类型：

```
>>> isinstance(h, Animal)
True
```

同理，实际类型是Dog的 `d` 也是Animal类型：

```
>>> isinstance(d, Dog) and isinstance(d, Animal)
True
```

但是， `d` 不是Husky类型：

```
>>> isinstance(d, Husky)
False
```

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断：

```
>>> isinstance('a', str)
True
>>> isinstance(u'a', unicode)
True
>>> isinstance('a', unicode)
False
```

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是str或者unicode：

```
>>> isinstance('a', (str, unicode))
True
>>> isinstance(u'a', (str, unicode))
True
```

由于 `str` 和 `unicode` 都是从 `basestring` 继承下来的，所以，还可以把上面的代码简化为：

```
>>> isinstance(u'a', basestring)
True
```

使用dir()

如果要获得一个对象的所有属性和方法，可以使用 `dir()` 函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
```

类似 `__xxx__` 的属性和方法在Python中都是有特殊用途的，比如 `__len__` 方法返回长度。在Python中，如果你调用 `len()` 函数试图获取一个对象的长度，实际上，在 `len()` 函数内部，它自动去调用该对象的 `__len__()` 方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

我们自己写的类，如果也想用 `len(myObj)` 的话，就自己写一个 `__len__()` 方法：

```
>>> class MyObject(object):
...     def __len__(self):
...         return 100
...
>>> obj = MyObject()
>>> len(obj)
100
```

剩下的都是普通属性或方法，比如 `lower()` 返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

仅仅把属性和方法列出来是不够的，配合 `getattr()` 、 `setattr()` 以及 `hasattr()` ，我们可以直接操作一个对象的状态：

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗？
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗？
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗？
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出AttributeError的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个default参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404
404
```

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗？
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x108c
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
>>> fn # fn指向obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x108c
>>> fn() # 调用fn()与调用obj.power()是一样的
81
```

小结

通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

```
sum = obj.x + obj.y
```

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。`hasattr()` 就派上了用场。

请注意，在Python这类动态语言中，有 `read()` 方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要 `read()` 方法返回的是有效的图像数据，就不影响读取图像的功能。

面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

使用__slots__

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

```
>>> class Student(object):
...     pass
...
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print s.name
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s, Student) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给class绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = MethodType(set_score, None, Student)
```

给class绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的 `set_score` 方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

使用slots

但是，如果我们想要限制class的属性怎么办？比如，只允许对Student实例添加 `name` 和 `age` 属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的 `__slots__` 变量，来限制该class能添加的属性：

```
>>> class Student(object):
...     __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
...
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于 `'score'` 没有被放到 `__slots__` 中，所以不能绑定 `score` 属性，试图绑定 `score` 将得到AttributeError的错误。

使用 `__slots__` 要注意， `__slots__` 定义的属性仅对当前类起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义 `__slots__` ，这样，子类允许定义的属性就是自身的 `__slots__` 加上父类的 `__slots__` 。

使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()
s.score = 9999
```

这显然不合逻辑。为了限制score的范围，可以通过一个 `set_score()` 方法来设置成绩，再通过一个 `get_score()` 来获取成绩，这样，在 `set_score()` 方法里，就可以检查参数：

```
class Student(object):

    def get_score(self):
        return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的 `@property` 装饰器就是负责把一个方法变成属性调用的：

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

`@property` 的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上 `@property` 就可以了，此时， `@property` 本身又创建了另一个装饰器 `@score.setter` ，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK，实际转化为s.set_score(60)
>>> s.score # OK，实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的 `@property` ，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2014 - self._birth
```

上面的 `birth` 是可读写属性，而 `age` 就是一个只读属性，因为 `age` 可以根据 `birth` 和当前时间计算出来。

小结

`@property` 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

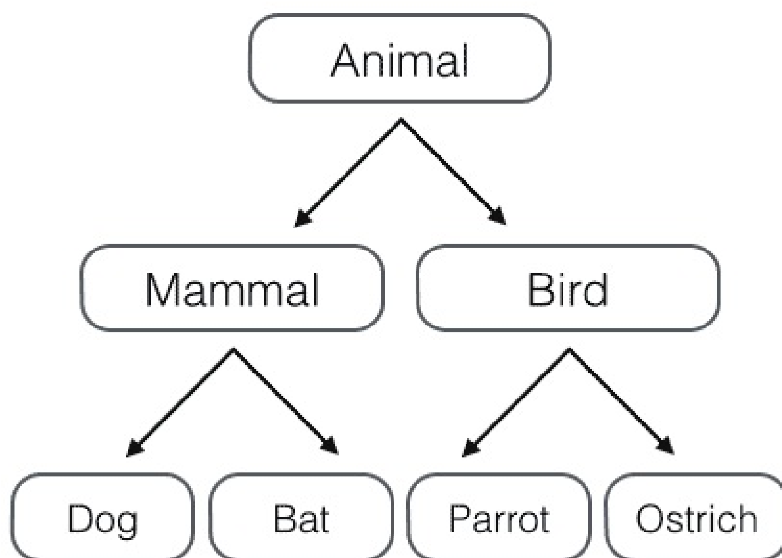
多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

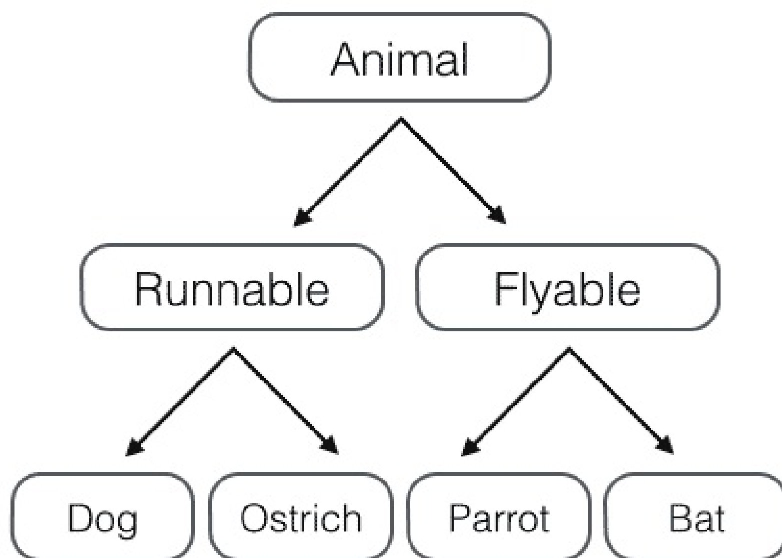
回忆一下 `Animal` 类层次的设计，假设我们要实现以下4种动物：

- Dog - 狗狗；
- Bat - 蝙蝠；
- Parrot - 鹦鹉；
- Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



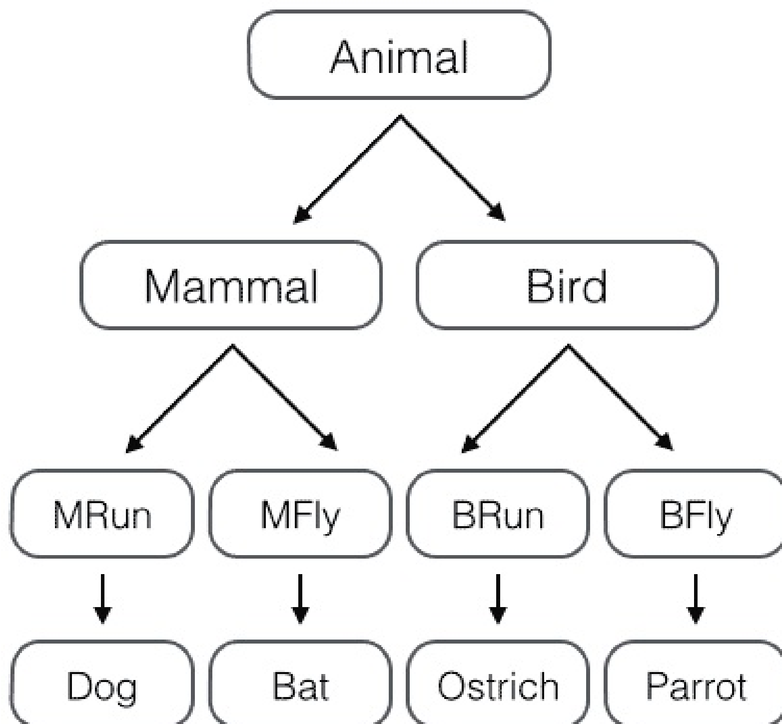
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

- 哺乳类：能跑的哺乳类，能飞的哺乳类；
- 鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

```
class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass

class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass

class Bat(Mammal):
    pass

class Parrot(Bird):
    pass

class Ostrich(Bird):
    pass
```

现在，我们要给动物再加上 `Runnable` 和 `Flyable` 的功能，只需要先定义好 `Runnable` 和 `Flyable` 的类：

```
class Runnable(object):
    def run(self):
        print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')
```

对于需要 `Runnable` 功能的动物，就多继承一个 `Runnable` ，例如 `Dog` ：

```
class Dog(Mammal, Runnable):
    pass
```

对于需要 `Flyable` 功能的动物，就多继承一个 `Flyable` ，例如 `Bat` ：

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

Mixin

在设计类的继承关系时，通常，主线都是单一继承下来的，例如， `Ostrich` 继承自 `Bird` 。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让 `Ostrich` 除了继承自 `Bird` 外，再同时继承 `Runnable` 。这种设计通常称之为Mixin。

为了更好地看出继承关系，我们把 `Runnable` 和 `Flyable` 改为 `RunnableMixin` 和 `FlyableMixin` 。类似的，你还可以定义出肉食动物 `CarnivorousMixin` 和植食动物 `HerbivoresMixin` ，让某个动物同时拥有好几个Mixin：

```
class Dog(Mammal, RunnableMixin, CarnivorousMixin):
    pass
```

Mixin的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个Mixin的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了Mixin。举个例子，Python自带了 `TCPServer` 和 `UDPServer` 这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由 `ForkingMixin` 和 `ThreadingMixin` 提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

```
class MyTCPServer(TCPServer, ForkingMixin):
    pass
```

编写一个多线程模式的UDP服务，定义如下：

```
class MyUDPServer(UDPServer, ThreadingMixin):
    pass
```

如果你打算搞一个更先进的协程模型，可以编写一个 `CoroutineMixin` ：

```
class MyTCPServer(TCPServer, CoroutineMixin):
    pass
```

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

小结

由于Python允许使用多重继承，因此，Mixin就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用Mixin的设计。

定制类

看到类似 `__slots__` 这种形如 `__xxx__` 的变量或者函数名就要注意，这些在Python中是有特殊用途的。

`__slots__` 我们已经知道怎么用了，`__len__()` 方法我们也知道是为了能让class作用于 `len()` 函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

str

我们先定义一个 `Student` 类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print Student('Michael')
<__main__.Student object at 0x109afb190>
```

打印出一堆 `<__main__.Student object at 0x109afb190>` ，不好看。

怎么才能打印得好看呢？只需要定义好 `__str__()` 方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print Student('Michael')
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用 `print` ，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是 `__str__()` ，而是 `__repr__()` ，两者的区别是 `__str__()` 返回用户看到的字符串，而 `__repr__()` 返回程序开发者看到的字符串，也就是说， `__repr__()` 是为调试服务的。

解决办法是再定义一个 `__repr__()` 。但是通常 `__str__()` 和 `__repr__()` 代码都是一样的，所以，有个偷懒的写法：

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

iter

如果一个类想被用于 `for ... in` 循环，类似list或tuple那样，就必须实现一个 `__iter__()` 方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的 `next()` 方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def next(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration();
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print n
...
1
1
2
3
5
...
46368
75025
```

getitem

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

```
>>> Fib()[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Fib' object does not support indexing
```

要表现得像list那样按照下标取出元素，需要实现 `__getitem__()` 方法：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

但是list有个神奇的切片方法：

```
>>> range(100)[5:10]
[5, 6, 7, 8, 9]
```

对于Fib却报错。原因是 `__getitem__()` 传入的参数可能是一个int，也可能是一个切片对象 `slice` ，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int):
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice):
            start = n.start
            stop = n.stop
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

现在试试Fib的切片：

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理：

```
>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

也没有对负数作处理，所以，要正确实现一个 `__getitem__()` 还是有很多工作要做的。

此外，如果把对象看成 `dict` ， `__getitem__()` 的参数也可能是一个可以作key的object，例如 `str` 。

与之对应的是 `__setitem__()` 方法，把对象视作list或dict来对集合赋值。最后，还有一个 `__delitem__()` 方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

getattr

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义 `Student` 类：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'
```

调用 `name` 属性，没问题，但是，调用不存在的 `score` 属性，就有问题了：

```
>>> s = Student()
>>> print s.name
Michael
>>> print s.score
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'score'
```

错误信息很清楚地告诉我们，没有找到 `score` 这个attribute。

要避免这个错误，除了可以加上一个 `score` 属性外，Python还有另一个机制，那就是写一个 `__getattr__()` 方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如 `score` ，Python解释器会试图调用 `__getattr__(self, 'score')` 来尝试获得属性，这样，我们就有机会返回 `score` 的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用 `__getattr__` ，已有的属性，比如 `name` ，不会在 `__getattr__` 中查找。

此外，注意到任意调用如 `s.abc` 都会返回 `None` ，这是因为我们定义的 `__getattr__` 默认返回就是 `None` 。要让class只响应特定的几个属性，我们就要按照约定，抛出 `AttributeError` 的错误：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no attribute \
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

```
http://api.server/user/friends
http://api.server/user/timeline/list
```

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的 `__getattr__` ，我们可以写出一个链式调用：

```
class Chain(object):

    def __init__(self, path=''):
        self._path = path

    def __getattr__(self, path):
        return Chain('%s/%s' % (self._path, path))

    def __str__(self):
        return self._path
```

试试：

```
>>> Chain().status.user.timeline.list
'/status/user/timeline/list'
```

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

```
GET /users/:user/repos
```

调用时，需要把 `:user` 替换为实际用户名。如果我们能写出这样的链式调用：

```
Chain().users('michael').repos
```

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

call

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用 `instance.method()` 来调用。能不能直接在实例本身上调用呢？类似 `instance()` ？在Python中，答案是肯定的。

任何类，只需要定义一个 `__call__()` 方法，就可以直接对实例进行调用。请看示例：

```
class Student(object):
    def __init__(self, name):
        self.name = name

    def __call__(self):
        print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('Michael')
>>> s()
My name is Michael.
```

`__call__()` 还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个 `Callable` 对象，比如函数和我们上面定义的带有 `__call()__` 的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('string')
False
```

通过 `callable()` 函数，我们就可以判断一个对象是否是“可调用”对象。

小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考Python的官方文档。

使用元类

type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个 `Hello` 的class，就写一个 `hello.py` 模块：

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

当Python解释器载入 `hello` 模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个 `Hello` 的class对象，测试如下：

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<type 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

`type()` 函数可以查看一个类型或变量的类型， `Hello` 是一个class，它的类型就是 `type` ，而 `h` 是一个实例，它的类型就是class `Hello` 。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用 `type()` 函数。

`type()` 函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过 `type()` 函数创建出 `Hello` 类，而无需通过 `class Hello(object)...` 的定义：

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello cl
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<type 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

要创建一个class对象，`type()`函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数`fn`绑定到方法名`hello`上。

通过`type()`函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class。

正常情况下，我们都用`class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

metaclass

除了使用`type()`动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个 `add` 方法：

定义 `ListMetaclass` ，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

```
# metaclass是创建类，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)

class MyList(list):
    __metaclass__ = ListMetaclass # 指示使用ListMetaclass来定制类
```

当我们写下 `__metaclass__ = ListMetaclass` 语句时，魔术就生效了，它指示Python解释器在创建 `MyList` 时，要通过 `ListMetaclass.__new__()` 来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

`__new__()` 方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合。

测试一下 `MyList` 是否可以调用 `add()` 方法：

```
>>> L = MyList()
>>> L.add(1)
>>> L
[1]
```

而普通的 `list` 没有 `add()` 方法：

```
>>> l = list()
>>> l.add(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'
```

动态修改有什么意义？直接在 `MyList` 定义中写上 `add()` 方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个 `User` 类来操作对应的数据库表 `User` ，我们期待他写出这样的代码：

```
class User(Model):
    # 定义类的属性到列的映射：
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例：
u = User(id=12345, name='Michael', email='test@orm.org', password=
# 保存到数据库：
u.save()
```

其中，父类 `Model` 和属性类型 `StringField` 、 `IntegerField` 是由ORM框架提供的，剩下的魔术方法比如 `save()` 全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义 `Field` 类，它负责保存数据库表的字段名和字段类型：

```
class Field(object):
    def __init__(self, name, column_type):
        self.name = name
        self.column_type = column_type
    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__, self.name)
```

在 `Field` 的基础上，进一步定义各种类型的 `Field` ，比如 `StringField` ， `IntegerField` 等等：

```
class StringField(Field):
    def __init__(self, name):
        super(StringField, self).__init__(name, 'varchar(100)')

class IntegerField(Field):
    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')
```

下一步，就是编写最复杂的 `ModelMetaclass` 了：

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        mappings = dict()
        for k, v in attrs.iteritems():
            if isinstance(v, Field):
                print('Found mapping: %s==>%s' % (k, v))
                mappings[k] = v
        for k in mappings.iterkeys():
            attrs.pop(k)
        attrs['__table__'] = name # 假设表名和类名一致
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        return type.__new__(cls, name, bases, attrs)
```

以及基类 `Model` ：

```
class Model(dict):
    __metaclass__ = ModelMetaclass

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.iteritems():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__,
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

当用户定义一个 `class User(Model)` 时，Python解释器首先在当前类 `User` 的定义中查找 `__metaclass__` ，如果没有找到，就继续在父类 `Model` 中查找 `__metaclass__` ，找到了，就使用 `Model` 中定义的 `__metaclass__` 的 `ModelMetaclass` 来创建 `User` 类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在 `ModelMetaclass` 中，一共做了几件事情：

1. 排除掉对 `Model` 类的修改；

2. 在当前类（比如 `User` ）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个 `__mappings__` 的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误；

3. 把表名保存到 `__table__` 中，这里简化为表名默认为类名。

在 `Model` 类中，就可以定义各种操作数据库的方法，比如 `save()` ， `delete()` ， `find()` ， `update` 等等。

我们实现了 `save()` 方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出 `INSERT` 语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org', password=
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,uid) values (?,?,?,?
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
```

可以看到， `save()` 方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架，完整的代码从这里下载：

https://github.com/michaelliao/learn-python/blob/master/metaclass/simple_orm.py

最后解释一下类属性和实例属性。直接在class中定义的是类属性：

```
class Student(object):
    name = 'Student'
```

实例属性必须通过实例来绑定，比如 `self.name = 'xxx'` 。来测试一下：

```
>>> # 创建实例s：
>>> s = Student()
>>> # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性：
>>> print(s.name)
Student
>>> # 这和调用Student.name是一样的：
>>> print(Student.name)
Student
>>> # 给实例绑定name属性：
>>> s.name = 'Michael'
>>> # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性：
>>> print(s.name)
Michael
>>> # 但是类属性并未消失，用Student.name仍然可以访问：
>>> print(Student.name)
Student
>>> # 如果删除实例的name属性：
>>> del s.name
>>> # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了：
>>> print(s.name)
Student
```

因此，在编写程序的时候，千万不要把实例属性和类属性使用相同的名字。

在我们编写的ORM中， `ModelMetaclass` 会删除掉User类的所有类属性，目的就是避免造成混淆。

错误、调试和测试

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数 `open()` ，成功时返回文件描述符（就是一个整数），出错时返回 `-1` 。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print 'Error'
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python也不例外。

try

让我们用一个例子来看看 `try` 的机制：

```
try:
    print 'try...'
    r = 10 / 0
    print 'result:', r
except ZeroDivisionError, e:
    print 'except:', e
finally:
    print 'finally...'
print 'END'
```

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块，至此，执行完毕。

上面的代码在计算 `10 / 0` 时会产生一个除法运算错误：

```
try...
except: integer division or modulo by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句 `print 'result:', r` 不会被执行， `except` 由于捕获到 `ZeroDivisionError` ，因此被执行。最后， `finally` 语句被执行。然后，程序继续按照流程往下走。

如果把除数 `0` 改成 `2` ，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以 `except` 语句块不会被执行，但是 `finally` 如果有，则一定会被执行（可以没有 `finally` 语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 `except` 语句块处理。没错，可以有多个 `except` 来捕获不同类型的错误：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
finally:
    print 'finally...'
print 'END'
```

`int()` 函数可能会抛出 `ValueError` ，所以我们用一个 `except` 捕获 `ValueError` ，用另一个 `except` 捕获 `ZeroDivisionError` 。

此外，如果没有错误发生，可以在 `except` 语句块后面加一个 `else` ，当没有错误发生时，会自动执行 `else` 语句：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
else:
    print 'no error!'
finally:
    print 'finally...'
print 'END'
```

Python的错误其实也是class，所有的错误类型都继承自 `BaseException` ，所以在使用 `except` 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except StandardError, e:
    print 'StandardError'
except ValueError, e:
    print 'ValueError'
```

第二个 `except` 永远也捕获不到 `ValueError` ，因为 `ValueError` 是 `StandardError` 的子类，如果有，也被第一个 `except` 给捕获了。

Python所有的错误都是从 `BaseException` 类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/2/library/exceptions.html#exception-hierarchy

使用 `try...except` 捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数 `main()` 调用 `foo()` ， `foo()` 调用 `bar()` ，结果 `bar()` 出错了，这时，只要 `main()` 捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        print 'Error!'
    finally:
        print 'finally...'
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 `try...except...finally` 的麻烦。

调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看 `err.py` ：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2行：

```
  File "err.py", line 11, in <module>
    main()
```

调用 `main()` 出错了，在代码文件 `err.py` 的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用 `bar('0')` 出错了，在代码文件 `err.py` 的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是 `return foo(s) * 2` 这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是 `return 10 / int(s)` 这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型 `ZeroDivisionError` ，我们判断， `int(s)` 本身并没有出错，但是 `int(s)` 返回 `0` ，在计算 `10 / 0` 时出错，至此，找到错误源头。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的 `logging` 模块可以非常容易地记录错误信息：

```
# err.py
import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        logging.exception(e)

main()
print 'END'
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python err.py
ERROR:root:integer division or modulo by zero
Traceback (most recent call last):
  File "err.py", line 12, in main
    bar('0')
  File "err.py", line 8, in bar
    return foo(s) * 2
  File "err.py", line 5, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
END
```

通过配置， `logging` 还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用 `raise` 语句抛出一个错误的实例：

```
# err.py
class FooError(StandardError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python err.py
Traceback (most recent call last):
  ...
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如ValueError，TypeError），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err.py
def foo(s):
    n = int(s)
    return 10 / n

def bar(s):
    try:
        return foo(s) * 2
    except StandardError, e:
        print 'Error!'
        raise

def main():
    bar('0')

main()
```

在 `bar()` 函数中，我们明明已经捕获了错误，但是，打印一个 `Error!` 后，又把错误通过 `raise` 语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。

`raise` 语句如果不带参数，就会把当前错误原样抛出。此外，在 `except` 中 `raise` 一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 `IOError` 转换成毫不相干的 `ValueError` 。

小结

Python内置的 `try...except...finally` 用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用 `print` 把可能有问题的变量打印出来看看：

```
# err.py
def foo(s):
    n = int(s)
    print '>>> n = %d' % n
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用 `print` 最大的坏处是将来还得删掉它，想想程序里到处都是 `print` ，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用 `print` 来辅助查看的地方，都可以用断言（assert）来替代：

```
# err.py
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert` 的意思是，表达式 `n != 0` 应该是 `True` ，否则，后面的代码就会出错。

如果断言失败， `assert` 语句本身就会抛出 `AssertionError` ：

```
$ python err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着 `assert` ，和 `print` 相比也好不到哪去。不过，启动Python解释器时可以用 `-O` 参数来关闭 `assert` ：

```
$ python -O err.py
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

关闭后，你可以把所有的 `assert` 语句当成 `pass` 来看。

logging

把 `print` 替换为 `logging` 是第3种方式，和 `assert` 比， `logging` 不会抛出错误，而且可以输出到文件：

```
# err.py
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print 10 / n
```

`logging.info()` 就可以输出一段文本。运行，发现除了 `ZeroDivisionError` ，没有任何信息。怎么回事？

别急，在 `import logging` 之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这就是 `logging` 的好处，它允许你指定记录信息的级别，有 `debug` ， `info` ， `warning` ， `error` 等几个级别，当我们指定 `level=INFO` 时， `logging.debug` 就不起作用了。同理，指定 `level=WARNING` 后， `debug` 和 `info` 就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging` 的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print 10 / n
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/sicp/err.py(2)<module>()
-> s = '0'
```

以参数 `-m pdb` 启动后，pdb定位到下一步要执行的代码 `-> s = '0'` 。输入命令 `l` 来查看代码：

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print 10 / n
[EOF]
```

输入命令 `n` 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/sicp/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
```

任何时候都可以输入命令 `p 变量名` 来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令 `q` 结束调试，退出程序：

```
(Pdb) n
ZeroDivisionError: 'integer division or modulo by zero'
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

pdb.set_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要 `import pdb` ，然后，在可能出错的地方放一个 `pdb.set_trace()` ，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print 10 / n
```

运行代码，程序会自动在 `pdb.set_trace()` 暂停并进入pdb调试环境，可以用命令 `p` 查看变量，或者用命令 `c` 继续运行：

```
$ python err.py
> /Users/michael/Github/sicp/err.py(7)<module>()
-> print 10 / n
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有PyCharm：

http://www.jetbrains.com/pycharm/

另外，Eclipse加上pydev插件也可以调试Python程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数 `abs()`，我们可以编写出以下几个测试用例：

1. 输入正数，比如 `1`、`1.2`、`0.99`，期待返回值与输入相同；
2. 输入负数，比如 `-1`、`-1.2`、`-0.99`，期待返回值与输入相反；
3. 输入 `0`，期待返回 `0`；
4. 输入非数值类型，比如 `None`、`[]`、`{}`，期待抛出 `TypeError`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对 `abs()` 函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对 `abs()` 函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个 `Dict` 类，这个类的行为和 `dict` 一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py` 代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的 `unittest` 模块，编写 `mydict_test.py` 如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEquals(d.a, 1)
        self.assertEquals(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEquals(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEquals(d['key'], 'value')

    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEquals()` ：

```
self.assertEquals(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过 `d['empty']` 访问不存在的key时，断言会抛出 `KeyError` ：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过 `d.empty` 访问不存在的key时，我们期待抛出 `AttributeError` ：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在 `mydict_test.py` 的最后加上两行代码：

```
if __name__ == '__main__':
    unittest.main()
```

这样就可以把 `mydict_test.py` 当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种更常见的方法是在命令行通过参数 `-m unittest` 直接运行单元测试：

```
$ python -m unittest mydict_test
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的 `setUp()` 和 `tearDown()` 方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()` 和 `tearDown()` 方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 `setUp()` 方法中连接数据库，在 `tearDown()` 方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print 'setUp...'

    def tearDown(self):
        print 'tearDown...'
```

可以再次运行测试看看每个测试方法调用前后是否会打印出 `setUp...` 和 `tearDown...` 。

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

文档测试

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如re模块就带了很多示例代码：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用`...`表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的`Dict`类：

```
class Dict(dict):
    '''
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
        ...
    KeyError: 'empty'
    >>> d2.empty
    Traceback (most recent call last):
        ...
    AttributeError: 'Dict' object has no attribute 'empty'
    '''
    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value

if __name__=='__main__':
    import doctest
    doctest.testmod()
```

运行 `python mydict.py` ：

```
$ python mydict.py
```

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把 `__getattr__()` 方法注释掉，再运行就会报错：

```
$ python mydict.py
*********************************************************************
File "mydict.py", line 7, in __main__.Dict
Failed example:
    d1.x
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'x'
*********************************************************************
File "mydict.py", line 13, in __main__.Dict
Failed example:
    d2.c
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'c'
*********************************************************************
```

注意到最后两行代码。当模块正常导入时，doctest不会被执行。只有在命令行运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

小结

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest。

IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在， `open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '/Users/michael/notfo
```

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 `IOError` ，一旦出错，后面的 `f.close()` 就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

```
try:
    f = open('/path/to/file', 'r')
    print f.read()
finally:
    if f:
        f.close()
```

但是每次都这么写实在太繁琐，所以，Python引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```
with open('/path/to/file', 'r') as f:
    print f.read()
```

这和前面的 `try ... finally` 是一样的，但是代码更佳简洁，并且不必调用 `f.close()` 方法。

调用 `read()` 会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取size个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list` 。因此，要根据需要决定怎么调用。

如果文件很小， `read()` 一次性读取最方便；如果不能确定文件大小，反复调用 `read(size)` 比较保险；如果是配置文件，调用 `readlines()` 最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像 `open()` 函数返回的这种有个 `read()` 方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个 `read()` 方法就行。

`StringIO` 就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是ASCII编码的文本文件。要读取二进制文件，比如图片、视频等等，用 `'rb'` 模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

字符编码

要读取非ASCII编码的文本文件，就必须以二进制模式打开，再解码。比如GBK编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'rb')
>>> u = f.read().decode('gbk')
>>> u
u'\u6d4b\u8bd5'
>>> print u
测试
```

如果每次都这么手动转换编码嫌麻烦（写程序怕麻烦是好事，不怕麻烦就会写出又长又难懂又没法维护的代码），Python还提供了一个 `codecs` 模块帮我们在读文件时自动转换编码，直接读出unicode：

```
import codecs
with codecs.open('/Users/michael/gbk.txt', 'r', 'gbk') as f:
    f.read() # u'\u6d4b\u8bd5'
```

写文件

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，传入标识符 `'w'` 或者 `'wb'` 表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用 `write()` 来写入文件，但是务必要调用 `f.close()` 来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 `close()` 方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用 `with` 语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请效仿 `codecs` 的示例，写入unicode，由 `codecs` 自动转换成指定编码。

小结

在Python中，文件读写是通过 `open()` 函数打开的文件对象完成的。使用 `with` 语句操作文件IO是个好习惯。

操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如 `dir` 、 `cp` 等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的 `os` 模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用 `os` 模块的基本功能：

```
>>> import os
>>> os.name # 操作系统名字
'posix'
```

如果是 `posix` ，说明系统是 `Linux` 、 `Unix` 或 `Mac OS X` ，如果是 `nt` ，就是 `Windows` 系统。

要获取详细的系统信息，可以调用 `uname()` 函数：

```
>>> os.uname()
('Darwin', 'iMac.local', '13.3.0', 'Darwin Kernel Version 13.3.0: T
```



注意 `uname()` 函数在Windows上不提供，也就是说，os模块的某些函数是跟操作系统相关的。

环境变量

在操作系统中定义的环境变量，全部保存在 `os.environ` 这个 `dict` 中，可以直接查看：

```
>>> os.environ
{'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'TERM_PROGRAM_VERSION': '3
```



要获取某个环境变量的值，可以调用 `os.getenv()` 函数：

```
>>> os.getenv('PATH')
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/loc
```

操作文件和目录

操作文件和目录的函数一部分放在 `os` 模块中，一部分放在 `os.path` 模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

```
# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Users/michael'
# 在某个目录下创建一个新目录，
# 首先把新目录的完整路径表示出来:
>>> os.path.join('/Users/michael', 'testdir')
'/Users/michael/testdir'
# 然后创建一个目录:
>>> os.mkdir('/Users/michael/testdir')
# 删掉一个目录:
>>> os.rmdir('/Users/michael/testdir')
```

把两个路径合成一个时，不要直接拼字符串，而要通过 `os.path.join()` 函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下， `os.path.join()` 返回这样的字符串：

```
part-1/part-2
```

而Windows下会返回这样的字符串：

```
part-1\part-2
```

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过 `os.path.split()` 函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

`os.path.splitext()` 可以直接让你得到文件扩展名，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个 `test.txt` 文件：

```
# 对文件重命名:
>>> os.rename('test.txt', 'test.py')
# 删掉文件:
>>> os.remove('test.py')
```

但是复制文件的函数居然在 `os` 模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是 `shutil` 模块提供了 `copyfile()` 的函数，你还可以在 `shutil` 模块中找到很多实用函数，它们可以看做是 `os` 模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Adlm
```

要列出所有的 `.py` 文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.sp
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py',
```

是不是非常简洁？

小结

Python的 `os` 模块封装了操作系统的目录和文件操作，要注意这些函数有的在 `os` 模块中，有的在 `os.path` 模块中。

练习：编写一个 `search(s)` 的函数，能在当前目录以及当前目录的所有子目录下查找文件名包含指定字符串的文件，并打印出完整路径：

```
$ python search.py test
unit_test.log
py/test.py
py/test_os.py
my/logs/unit-test-result.txt
```

序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把 `name` 改成 `'Bill'` ，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的 `'Bill'` 存储到磁盘上，下次重新运行程序，变量又被初始化为 `'Bob'` 。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供两个模块来实现序列化： `cPickle` 和 `pickle` 。这两个模块功能是一样的，区别在于 `cPickle` 是C语言写的，速度快， `pickle` 是纯Python写的，速度慢，跟 `cStringIO` 和 `StringIO` 一个道理。用的时候，先尝试导入 `cPickle` ，如果失败，再导入 `pickle` ：

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

首先，我们尝试把一个对象序列化并写入文件：

```
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
"(dp0\nS'age'\np1\nI20\nsS'score'\np2\nI88\nsS'name'\np3\nS'Bob'\np
```

`pickle.dumps()` 方法把任意对象序列化成一个str，然后，就可以把这个str写入文件。或者用另一个方法 `pickle.dump()` 直接把对象序列化后写入一个file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 `dump.txt` 文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个 `str` ，然后用 `pickle.loads()` 方法反序列化出对象，也可以直接用 `pickle.load()` 方法从一个 `file-like Object` 中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	Python类型
{}	dict
[]	list
"string"	'str'或u'unicode'
1234.56	int或float
true/false	True/False
null	None

Python内置的 `json` 模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`dumps()` 方法返回一个 `str` ，内容就是标准的JSON。类似的， `dump()` 方法可以直接把JSON写入一个 `file-like Object` 。

要把JSON反序列化为Python对象，用 `loads()` 或者对应的 `load()` 方法，前者把JSON的字符串反序列化，后者从 `file-like Object` 中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{u'age': 20, u'score': 88, u'name': u'Bob'}
```

有一点需要注意，就是反序列化得到的所有字符串对象默认都是 `unicode` 而不是 `str` 。由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的 `str` 或 `unicode` 与JSON的字符串之间转换。

JSON进阶

Python的 `dict` 对象可以直接序列化为JSON的 `{}` ，不过，很多时候，我们更喜欢用 `class` 表示对象，比如定义 `Student` 类，然后序列化：

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 `TypeError` ：

```
Traceback (most recent call last):
  ...
TypeError: <__main__.Student object at 0x10aabef50> is not JSON ser
```

错误的原因是 `Student` 对象不是一个可序列化为JSON的对象。

如果连 `class` 的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看 `dumps()` 方法的参数列表，可以发现，除了第一个必须的 `obj` 参数外， `dumps()` 方法还提供了一大堆的可选参数：

https://docs.python.org/2/library/json.html#json.dumps

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把 `Student` 类实例序列化为JSON，是因为默认情况下， `dumps()` 方法不知道如何将 `Student` 实例变为一个JSON的 `{}` 对象。

可选参数 `default` 就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为 `Student` 专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }

print(json.dumps(s, default=student2dict))
```

这样， `Student` 实例首先被 `student2dict()` 函数转换成 `dict` ，然后再被顺利序列化为JSON。

不过，下次如果遇到一个 `Teacher` 类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意 `class` 的实例变为 `dict` ：

```
print(json.dumps(s, default=lambda obj: obj.__dict__))
```

因为通常 `class` 的实例都有一个 `__dict__` 属性，它就是一个 `dict` ，用来存储实例变量。也有少数例外，比如定义了 `__slots__` 的class。

同样的道理，如果我们要把JSON反序列化为一个 `Student` 对象实例， `loads()` 方法首先转换出一个 `dict` 对象，然后，我们传入的 `object_hook` 函数负责把 `dict` 转换为 `Student` 实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])

json_str = '{"age": 20, "score": 88, "name": "Bob"}'
print(json.loads(json_str, object_hook=dict2student))
```

运行结果如下：

```
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 `Student` 实例对象。

小结

Python语言特定的序列化模块是 `pickle` ，但如果要把序列化搞得更通用、更符合Web标准，就可以使用 `json` 模块。

`json` 模块的 `dumps()` 和 `loads()` 函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核CPU才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个 `fork()` 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是 `fork()` 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回 `0` ，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用 `getppid()` 就可以拿到父进程的ID。

Python的 `os` 模块封装了常见的系统调用，其中就包括 `fork` ，可以在Python程序中轻松创建子进程：

```
# multiprocessing.py
import os

print 'Process (%s) start...' % os.getpid()
pid = os.fork()
if pid==0:
    print 'I am child process (%s) and my parent is %s.' % (os.getp
else:
    print 'I (%s) just created a child process (%s).' % (os.getpid(
```

运行结果如下：

```
Process (876) start...
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

由于Windows没有 `fork` 调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了 `fork` 调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有 `fork` 调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。 `multiprocessing` 模块就是跨平台版本的多进程模块。

`multiprocessing` 模块提供了一个 `Process` 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print 'Run child process %s (%s)...' % (name, os.getpid())

if __name__=='__main__':
    print 'Parent process %s.' % os.getpid()
    p = Process(target=run_proc, args=('test',))
    print 'Process will start.'
    p.start()
    p.join()
    print 'Process end.'
```

执行结果如下：

```
Parent process 928.
Process will start.
Run child process test (929)...
Process end.
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 `Process` 实例，用 `start()` 方法启动，这样创建进程比 `fork()` 还要简单。

`join()` 方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print 'Run task %s (%s)...' % (name, os.getpid())
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print 'Task %s runs %0.2f seconds.' % (name, (end - start))

if __name__=='__main__':
    print 'Parent process %s.' % os.getpid()
    p = Pool()
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print 'Waiting for all subprocesses done...'
    p.close()
    p.join()
    print 'All subprocesses done.'
```

执行结果如下：

```
Parent process 669.
Waiting for all subprocesses done...
Run task 0 (671)...
Run task 1 (672)...
Run task 2 (673)...
Run task 3 (674)...
Task 2 runs 0.14 seconds.
Run task 4 (673)...
Task 1 runs 0.27 seconds.
Task 3 runs 0.86 seconds.
Task 0 runs 1.41 seconds.
Task 4 runs 1.91 seconds.
All subprocesses done.
```

代码解读：

对 `Pool` 对象调用 `join()` 方法会等待所有子进程执行完毕，调用 `join()` 之前必须先调用 `close()` ，调用 `close()` 之后就不能继续添加新的 `Process` 了。

请注意输出的结果，task `0` ， `1` ， `2` ， `3` 是立刻执行的，而task `4` 要等待前面某个task完成后才执行，这是因为 `Pool` 的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是 `Pool` 有意设计的限制，并不是操作系统的限制。如果改成：

```
p = Pool(5)
```

就可以同时跑5个进程。

由于 `Pool` 的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

进程间通信

`Process` 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的 `multiprocessing` 模块包装了底层的机制，提供了 `Queue` 、 `Pipes` 等多种方式来交换数据。

我们以 `Queue` 为例，在父进程中创建两个子进程，一个往 `Queue` 里写数据，一个从 `Queue` 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    for value in ['A', 'B', 'C']:
        print 'Put %s to queue...' % value
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    while True:
        value = q.get(True)
        print 'Get %s from queue.' % value

if __name__=='__main__':
    # 父进程创建Queue，并传给各个子进程：
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

运行结果如下：

```
Put A to queue...
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

在Unix/Linux下，`multiprocessing`模块封装了`fork()`调用，使我们不需要关注`fork()`的细节。由于Windows没有`fork`调用，因此，`multiprocessing`需要“模拟”出`fork`的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果`multiprocessing`在Windows下调用失败了，要先考虑是不是pickle失败了。

小结

在Unix/Linux下，可以使用`fork()`调用实现多进程。

要实现跨平台的多进程，可以使用`multiprocessing`模块。

进程间通信是通过`Queue`、`Pipes`等实现的。

多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：`thread`和`threading`，`thread`是低级模块，`threading`是高级模块，对`thread`进行了封装。绝大多数情况下，我们只需要使用`threading`这个高级模块。

启动一个线程就是把一个函数传入并创建`Thread`实例，然后调用`start()`开始执行：

```
import time, threading

# 新线程执行的代码:
def loop():
    print 'thread %s is running...' % threading.current_thread().na
    n = 0
    while n < 5:
        n = n + 1
        print 'thread %s >>> %s' % (threading.current_thread().name
        time.sleep(1)
    print 'thread %s ended.' % threading.current_thread().name

print 'thread %s is running...' % threading.current_thread().name
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print 'thread %s ended.' % threading.current_thread().name
```

执行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的 `threading` 模块有个 `current_thread()` 函数，它永远返回当前线程的实例。主线程实例的名字叫 `MainThread` ，子线程的名字在创建时指定，我们用 `LoopThread` 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为 `Thread-1` ， `Thread-2` ……

Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

```
import time, threading

# 假定这是你的银行存款:
balance = 0

def change_it(n):
    # 先存后取，结果应该为0:
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print balance
```

我们定义了一个共享变量 `balance` ，初始值为 `0` ，并且启动两个线程，先存后取，理论上结果应该为 `0` ，但是，由于线程的调度是由操作系统决定的，当t1、t2交替执行时，只要循环次数足够多， `balance` 的结果就不一定是 `0` 了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

```
balance = balance + n
```

也分两步：

1. 计算 `balance + n` ，存入临时变量中；
2. 将临时变量的值赋给 `balance` 。

也就是可以看成：

```
x = balance + n
balance = x
```

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5
t1: balance = x1     # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1     # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2     # balance = 8
t2: x2 = balance - 8 # x2 = 8 - 8 = 0
t2: balance = x2     # balance = 0

结果 balance = 0
```

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

```
初始值 balance = 0

t1: x1 = balance + 5  # x1 = 0 + 5 = 5

t2: x2 = balance + 8  # x2 = 0 + 8 = 8
t2: balance = x2      # balance = 8

t1: balance = x1      # balance = 5
t1: x1 = balance - 5  # x1 = 5 - 5 = 0
t1: balance = x1      # balance = 0

t2: x2 = balance - 5  # x2 = 0 - 5 = -5
t2: balance = x2      # balance = -5

结果 balance = -5
```

究其原因，是因为修改 `balance` 需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改 `balance` 的时候，别的线程一定不能改。

如果我们要确保 `balance` 计算正确，就要给 `change_it()` 上一把锁，当某个线程开始执行 `change_it()` 时，我们说，该线程因为获得了锁，因此其他线程不能同时执行 `change_it()` ，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过 `threading.Lock()` 来实现：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

当多个线程同时执行 `lock.acquire()` 时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 `try...finally` 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

```
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

for i in range(multiprocessing.cpu_count()):
    t = threading.Thread(target=loop)
    t.start()
```

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有160%，也就是使用不到两核。

即使启动100个线程，使用率也就170%左右，仍然不到两核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局

锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
def process_student(name):
    std = Student(name)
    # std是局部变量，但是每个函数都要用它，因此必须传进去：
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的 `Student` 对象，不能共享。

如果用一个全局 `dict` 存放所有的 `Student` 对象，然后以 `thread` 自身作为 `key` 获得线程对应的 `Student` 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把std放到全局变量global_dict中：
    global_dict[threading.current_thread()] = std
    do_task_1()
    do_task_2()

def do_task_1():
    # 不传入std，而是根据当前线程查找：
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
    # 任何函数都可以查找出当前线程的std变量：
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 `std` 对象在每层函数中的传递问题，但是，每个函数获取 `std` 的代码有点丑。

有没有更简单的方式？

`ThreadLocal` 应运而生，不用查找 `dict` ， `ThreadLocal` 帮你自动做这件事：

```
import threading

# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    print 'Hello, %s (in %s)' % (local_school.student, threading.cu

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name
t2 = threading.Thread(target= process_thread, args=('Bob',), name=
t1.start()
t2.start()
t1.join()
t2.join()
```

执行结果：

```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 `local_school` 就是一个 `ThreadLocal` 对象，每个 `Thread` 对它都可以读写 `student` 属性，但互不影响。你可以把 `local_school` 看成全局变量，但每个属性如 `local_school.student` 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题， `ThreadLocal` 内部会处理。

可以理解为全局变量 `local_school` 是一个 `dict` ，不但可以用 `local_school.student` ，还可以绑定其他变量，如 `local_school.teacher` 等等。

`ThreadLocal` 最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用 `fork` 调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示："该程序执行了非法操作，即将关闭"，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单进程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的 `multiprocessing` 模块不但支持多进程，其中 `managers` 子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于 `managers` 模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过 `Queue` 通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的 `Queue` 可以继续使用，但是，通过 `managers` 模块把 `Queue` 通过网络暴露出去，就可以让其他机器的进程访问 `Queue` 了。

我们先看服务进程，服务进程负责启动 `Queue` ，把 `Queue` 注册到网络上，然后往 `Queue` 里面写入任务：

```
# taskmanager.py

import random, time, Queue
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = Queue.Queue()
# 接收结果的队列:
result_queue = Queue.Queue()

# 从BaseManager继承的QueueManager:
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:
QueueManager.register('get_task_queue', callable=lambda: task_queue
QueueManager.register('get_result_queue', callable=lambda: result_q
# 绑定端口5000, 设置验证码'abc':
manager = QueueManager(address=('', 5000), authkey='abc')
# 启动Queue:
manager.start()
# 获得通过网络访问的Queue对象:
task = manager.get_task_queue()
result = manager.get_result_queue()
# 放几个任务进去:
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)
# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
```

请注意，当我们在一台机器上写多进程程序时，创建的 `Queue` 可以直接拿来用，但是，在分布式多进程环境下，添加任务到 `Queue` 不可以直接对原始的 `task_queue` 进行操作，那样就绕过了 `QueueManager` 的封装，必须通过 `manager.get_task_queue()` 获得的 `Queue` 接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

```
# taskworker.py

import time, sys, Queue
from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器，也就是运行taskmanager.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与taskmanager.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey='abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务,并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动 `taskmanager.py` 服务进程：

```
$ python taskmanager.py
Put task 3411...
Put task 1605...
Put task 1398...
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...
Put task 7866...
Try get results...
```

taskmanager进程发送完任务后，开始等待 `result` 队列的结果。现在启动 `taskworker.py` 进程：

```
$ python taskworker.py 127.0.0.1
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

taskworker进程结束，在taskmanager进程中会继续打印出结果：

```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这个简单的Manager/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算 `n*n` 的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到 `taskworker.py` 中根本没有创建Queue的代码，所以，Queue对象存储在 `taskmanager.py` 进程中：



而 `Queue` 之所以能通过网络访问，就是通过 `QueueManager` 实现的。由于 `QueueManager` 管理的不止一个 `Queue` ，所以，要给每个 `Queue` 的网络调用接口起个名字，比如 `get_task_queue` 。

`authkey` 有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果 `taskworker.py` 的 `authkey` 和 `taskmanager.py` 的 `authkey` 不一致，肯定连接不上。

小结

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 `@` 前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 `\d` 可以匹配一个数字， `\w` 可以匹配一个字母或数字，所以：

- `'00\d'` 可以匹配 `'007'` ，但无法匹配 `'00A'` ；
- `'\d\d\d'` 可以匹配 `'010'` ；
- `'\w\w\d'` 可以匹配 `'py3'` ；

`.` 可以匹配任意字符，所以：

- `'py.'` 可以匹配 `'pyc'` 、 `'pyo'` 、 `'py!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括0个），用 `+` 表示至少一个字符，用 `?` 表示0个或1个字符，用 `{n}` 表示n个字符，用 `{n,m}` 表示n-m个字符：

来看一个复杂的例子： `\d{3}\s+\d{3,8}` 。

我们来从左到右解读一下：

1. `\d{3}` 表示匹配3个数字，例如 `'010'` ；

2. `\s` 可以匹配一个空格（也包括Tab等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '` ， `'  '` 等；

3. `\d{3,8}` 表示3-8个数字，例如 `'1234567'` 。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `'\'` 转义，所以，上面的正则是 `\d{3}\-\d{3,8}` 。

但是，仍然无法匹配 `'010 - 12345'` ，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z\_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z\_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100'` ， `'0_Z'` ， `'Py3000'` 等等；
- `[a-zA-Z\_][0-9a-zA-Z\_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- `[a-zA-Z\_][0-9a-zA-Z\_]{0, 19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `[P|p]ython` 可以匹配 `'Python'` 或者 `'python'` 。

`^` 表示行的开头， `^\d` 表示必须以数字开头。

`$` 表示行的结束， `\d$` 表示必须以数字结束。

你可能注意到了， `py` 也可以匹配 `'python'` ，但是加上 `^py$` 就变成了整行匹配，就只能匹配 `'py'` 了。

re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供 `re` 模块，包含所有正则表达式的功能。由于Python的字符串本身也用 `\` 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\-001'
```

因此我们强烈建议使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')
<_sre.SRE_Match object at 0x1026e18b8>
>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None` 。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print 'ok'
else:
    print 'failed'
```

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
>>> 'a b   c'.split(' ')
['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s+', 'a b   c')
['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入 `,` 试试：

```
>>> re.split(r'[\s\,]+', 'a,b, c  d')
['a', 'b', 'c', 'd']
```

再加入 `;` 试试：

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c  d')
['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object at 0x1026fb3e8>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。

注意到 `group(0)` 永远是原始字符串， `group(1)` 、 `group(2)` ……表示第1、2、……个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2|
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'
```

对于 `'2-30'` ， `'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0` ：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于`\d+`采用贪婪匹配，直接把后面的`0`全部匹配了，结果`0*`只能匹配空字符串了。

必须让`\d+`采用非贪婪匹配（也就是尽可能少匹配），才能把后面的`0`匹配出来，加个`?`就可以让`\d+`采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用：
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

```
someone@gmail.com
bill.gates@microsoft.com
```

版本二可以验证并提取出带名字的Email地址：

```
<Tom Paris> tom@voyager.org
```

常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

本章将介绍一些常用的内建模块。

collections

collections是Python内建的一个集合模块，提供了许多有用的集合类。

namedtuple

我们知道 `tuple` 可以表示不变集合，例如，一个点的二维坐标就可以表示成：

```
>>> p = (1, 2)
```

但是，看到 `(1, 2)` ，很难看出这个 `tuple` 是用来表示一个坐标的。

定义一个class又小题大做了，这时， `namedtuple` 就派上了用场：

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。

这样一来，我们用 `namedtuple` 可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的 `Point` 对象是 `tuple` 的一种子类：

```
>>> isinstance(p, Point)
True
>>> isinstance(p, tuple)
True
```

类似的，如果要用坐标和半径表示一个圆，也可以用 `namedtuple` 定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

deque

使用 `list` 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 `list` 是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

`deque` 除了实现list的 `append()` 和 `pop()` 外，还支持 `appendleft()` 和 `popleft()` ，这样就可以非常高效地往头部添加或删除元素。

defaultdict

使用 `dict` 时，如果引用的Key不存在，就会抛出 `KeyError` 。如果希望key不存在时，返回一个默认值，就可以用 `defaultdict` ：

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1存在
'abc'
>>> dd['key2'] # key2不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建 `defaultdict` 对象时传入。

除了在Key不存在时返回默认值， `defaultdict` 的其他行为跟 `dict` 是完全一样的。

OrderedDict

使用 `dict` 时，Key是无序的。在对 `dict` 做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用 `OrderedDict` ：

```
>>> from collections import OrderedDict
>>> d = dict([('a', 1), ('b', 2), ('c', 3)])
>>> d # dict的Key是无序的
{'a': 1, 'c': 3, 'b': 2}
>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> od # OrderedDict的Key是有序的
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

注意， `OrderedDict` 的Key会按照插入的顺序排列，不是Key本身排序：

```
>>> od = OrderedDict()
>>> od['z'] = 1
>>> od['y'] = 2
>>> od['x'] = 3
>>> od.keys() # 按照插入的Key的顺序返回
['z', 'y', 'x']
```

`OrderedDict` 可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

```
from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):

    def __init__(self, capacity):
        super(LastUpdatedOrderedDict, self).__init__()
        self._capacity = capacity

    def __setitem__(self, key, value):
        containsKey = 1 if key in self else 0
        if len(self) - containsKey >= self._capacity:
            last = self.popitem(last=False)
            print 'remove:', last
        if containsKey:
            del self[key]
            print 'set:', (key, value)
        else:
            print 'add:', (key, value)
        OrderedDict.__setitem__(self, key, value)
```

Counter

`Counter` 是一个简单的计数器，例如，统计字符出现的个数：

```
>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p
```

`Counter` 实际上也是 `dict` 的一个子类，上面的结果可以看出，字符 `'g'` 、 `'m'` 、 `'r'` 各出现了两次，其他字符各出现了一次。

小结

`collections` 模块提供了一些有用的集合类，可以根据需要选用。

base64

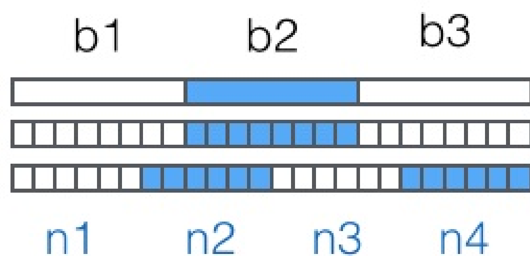
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开 `exe` 、 `jpg` 、 `pdf` 这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

```
['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']
```

然后，对二进制数据进行处理，每3个字节一组，一共是 `3x8=24` bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用 `\x00` 字节在末尾补足后，再在编码的末尾加上1个或2个 `=` 号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的 `base64` 可以直接进行base64的编解码：

```
>>> import base64
>>> base64.b64encode('binary\x00string')
'YmluYXJ5AHN0cmluZw=='
>>> base64.b64decode('YmluYXJ5AHN0cmluZw==')
'binary\x00string'
```

由于标准的Base64编码后可能出现字符 `+` 和 `/` ，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符 `+` 和 `/` 分别变成 `-` 和 `_` ：

```
>>> base64.b64encode('i\xb7\x1d\xfb\xef\xff')
'abcd++//'
>>> base64.urlsafe_b64encode('i\xb7\x1d\xfb\xef\xff')
'abcd--__'
>>> base64.urlsafe_b64decode('abcd--__')
'i\xb7\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于 `=` 字符也可能出现在Base64编码中，但 `=` 用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把 `=` 去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉 `=` 后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上 `=` 把Base64字符串的长度变为4的倍数，就可以正常解码了。

请写一个能处理去掉 `=` 的base64解码函数：

```
>>> base64.b64decode('YWJjZA==')
'abcd'
>>> base64.b64decode('YWJjZA')
Traceback (most recent call last):
  ...
TypeError: Incorrect padding
>>> safe_b64decode('YWJjZA')
'abcd'
```

小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

struct

准确地讲，Python没有专门处理字节的数据类型。但由于 `str` 既是字符串，又可以表示字节，所以，字节数组＝str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的str，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = chr((n & 0xff000000) >> 24)
>>> b2 = chr((n & 0xff0000) >> 16)
>>> b3 = chr((n & 0xff00) >> 8)
>>> b4 = chr(n & 0xff)
>>> s = b1 + b2 + b3 + b4
>>> s
'\x00\x9c@c'
```

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个 `struct` 模块来解决 `str` 和其他二进制数据类型的转换。

`struct` 的 `pack` 函数把任意数据类型变成字符串：

```
>>> import struct
>>> struct.pack('>I', 10240099)
'\x00\x9c@c'
```

`pack` 的第一个参数是处理指令， `'>I'` 的意思是：

`>` 表示字节顺序是big-endian，也就是网络序， `I` 表示4字节无符号整数。

后面的参数个数要和处理指令一致。

`unpack` 把 `str` 变成相应的数据类型：

```
>>> struct.unpack('>IH', '\xf0\xf0\xf0\xf0\x80\x80')
(4042322160, 32896)
```

根据 `>IH` 的说明，后面的 `str` 依次变为 `I` ：4字节无符号整数和 `H` ：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用 `struct` 就方便多了。

`struct` 模块定义的数据类型可以参考Python官方文档：

https://docs.python.org/2/library/struct.html#format-characters

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用 `struct` 分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

```
>>> s = '\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x
```

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节： `'BM'` 表示Windows位图， `'BA'` 表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用 `unpack` 读取：

```
>>> struct.unpack('<ccIIIIIIHH', s)
('B', 'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示， `'B'` 、 `'M'` 说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个 `bmpinfo.py` ，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

hashlib

摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串 `'how to use python hashlib - by Michael'` ，并附上这篇文章的摘要是 `'2d73d4f15c0db7f5ecb321b6a65e5d6d'` 。如果有人篡改了你的文章，并发表为 `'how to use python hashlib - by Bob'` ，你可以一下子指出Bob篡改了你的文章，因为根据 `'how to use python hashlib - by Bob'` 计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数 `f()` 对任意长度的数据 `data` 计算出固定长度的摘要 `digest` ，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算 `f(data)` 很容易，但通过 `digest` 反推 `data` 却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?')
print md5.hexdigest()
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用 `update()` ，最后计算的结果是一样的：

```
md5 = hashlib.md5()
md5.update('how to use md5 in ')
md5.update('python hashlib?')
print md5.hexdigest()
```

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in ')
sha1.update('python hashlib?')
print sha1.hexdigest()
```

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章 `'how to learn hashlib in python - by Bob'`，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

```
name    | password
--------+----------
michael | 123456
bob     | abc999
alice   | alice2008
```

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

```
username | password
---------+---------------------------------
michael  | e10adc3949ba59abbe56e057f20f883e
bob      | 878ef96e86145580c38c87f0410ad153
alice    | 99b1c2188db85afee403b1536010c2c9
```

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

练习：根据用户输入的口令，计算出存储在数据库中的MD5口令：

```
def calc_md5(password):
    pass
```

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

练习：设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

```
db = {
    'michael': 'e10adc3949ba59abbe56e057f20f883e',
    'bob': '878ef96e86145580c38c87f0410ad153',
    'alice': '99b1c2188db85afee403b1536010c2c9'
}

def login(user, password):
    pass
```

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用`123456`，`888888`，`password`这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

```
'e10adc3949ba59abbe56e057f20f883e': '123456'
'21218cca77804d2ba1922c33e0151105': '888888'
'5f4dcc3b5aa765d61d8327deb882cf99': 'password'
```

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

```
def calc_md5(password):
    return get_md5(password + 'the-Salt')
```

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如 `123456` ，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

练习：根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

```
db = {}

def register(username, password):
    db[username] = get_md5(password + username + 'the-Salt')
```

然后，根据修改后的MD5算法实现用户登录的验证：

```
def login(username, password):
    pass
```

小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

itertools

Python的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。

首先，我们看看 `itertools` 提供的几个“无限”迭代器：

```
>>> import itertools
>>> natuals = itertools.count(1)
>>> for n in natuals:
...     print n
...
1
2
3
...
```

因为 `count()` 会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按 `Ctrl+C` 退出。

`cycle()` 会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print c
...
'A'
'B'
'C'
'A'
'B'
'C'
...
```

同样停不下来。

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
>>> ns = itertools.repeat('A', 10)
>>> for n in ns:
...     print n
...
打印10次'A'
```

无限序列只有在 `for` 迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过 `takewhile()` 等函数根据条件判断来截取出一个有限的序列：

```
>>> natuals = itertools.count(1)
>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)
>>> for n in ns:
...     print n
...
打印出1到10
```

`itertools` 提供的几个迭代器操作函数更加有用：

chain()

`chain()` 可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
for c in chain('ABC', 'XYZ'):
    print c
# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'
```

groupby()

`groupby()` 把迭代器中相邻的重复元素挑出来放在一起：

```
>>> for key, group in itertools.groupby('AAABBBCCAAA'):
...     print key, list(group) # 为什么这里要用list()函数呢？
...
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素 `'A'` 和 `'a'` 都返回相同的key：

```
>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.
...     print key, list(group)
...
A ['A', 'a', 'a']
B ['B', 'B', 'b']
C ['c', 'C']
A ['A', 'A', 'a']
```

imap()

`imap()` 和 `map()` 的区别在于， `imap()` 可以作用于无穷序列，并且，如果两个序列的长度不一致，以短的那个为准。

```
>>> for x in itertools.imap(lambda x, y: x * y, [10, 20, 30], itert
...     print x
...
10
40
90
```

注意 `imap()` 返回一个迭代对象，而 `map()` 返回list。当你调用 `map()` 时，已经计算完毕：

```
>>> r = map(lambda x: x*x, [1, 2, 3])
>>> r # r已经计算出来了
[1, 4, 9]
```

当你调用 `imap()` 时，并没有进行任何计算：

```
>>> r = itertools.imap(lambda x: x*x, [1, 2, 3])
>>> r
<itertools.imap object at 0x103d3ff90>
# r只是一个迭代对象
```

必须用 `for` 循环对 `r` 进行迭代，才会在每次循环过程中计算出下一个元素：

```
>>> for x in r:
...     print x
...
1
4
9
```

这说明 `imap()` 实现了“惰性计算”，也就是在需要获得结果的时候才计算。类似 `imap()` 这样能够实现惰性计算的函数就可以处理无限序列：

```
>>> r = itertools.imap(lambda x: x*x, itertools.count(1))
>>> for n in itertools.takewhile(lambda x: x<100, r):
...     print n
...
结果是什么?
```

如果把 `imap()` 换成 `map()` 去处理无限序列会有什么结果？

```
>>> r = map(lambda x: x*x, itertools.count(1))
结果是什么?
```

ifilter()

不用多说了，`ifilter()`就是`filter()`的惰性实现。

小结

`itertools`模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是迭代对象，只有用`for`循环迭代的时候才真正计算。

XML

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是`start_element`，`end_element`和`char_data`，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

```
<a href="/">python</a>
```

会产生3个事件：

1. start_element事件，在读取`<a href="/">`时；
2. char_data事件，在读取`python`时；
3. end_element事件，在读取`</a>`时。

用代码实验一下：

```
from xml.parsers.expat import ParserCreate

class DefaultSaxHandler(object):
    def start_element(self, name, attrs):
        print('sax:start_element: %s, attrs: %s' % (name, str(attrs

    def end_element(self, name):
        print('sax:end_element: %s' % name)

    def char_data(self, text):
        print('sax:char_data: %s' % text)

xml = r'''<?xml version="1.0"?>
<ol>
    <li><a href="/python">Python</a></li>
    <li><a href="/ruby">Ruby</a></li>
</ol>
'''
handler = DefaultSaxHandler()
parser = ParserCreate()
parser.returns_unicode = True
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)
```

当设置 `returns_unicode` 为True时，返回的所有element名称和char_data都是unicode，处理国际化更方便。

需要注意的是读取一大段字符串时， `CharacterDataHandler` 可能被多次调用，所以需要自己保存起来，在 `EndElementHandler` 里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

```
L = []
L.append(r'<?xml version="1.0"?>')
L.append(r'<root>')
L.append(encode('some & data'))
L.append(r'</root>')
return ''.join(L)
```

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

练习一下解析Yahoo的XML格式的天气预报，获取当天和最近几天的天气：

```
http://weather.yahooapis.com/forecastrss?u=c&w=2151330
```

参数 `w` 是城市代码，要查询某个城市代码，可以在weather.yahoo.com搜索城市，浏览器地址栏的URL就包含城市代码。

HTMLParser

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

```
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print('<%s>' % tag)

    def handle_endtag(self, tag):
        print('</%s>' % tag)

    def handle_startendtag(self, tag, attrs):
        print('<%s/>' % tag)

    def handle_data(self, data):
        print('data')

    def handle_comment(self, data):
        print('<!-- -->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
        print('&#%s;' % name)

parser = MyHTMLParser()
parser.feed('<html><head></head><body><p>Some <a href=\"#\">html</a
```

`feed()` 方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的 ` ` ，一种是数字表示的 `Ӓ` ，这两种字符都可以通过Parser解析出来。

小结

找一个网页，例如https://www.python.org/events/python-events/，用浏览器查看源码并复制，然后尝试解析一下HTML，输出Python官网发布的会议时间、名称和地点。

常用第三方模块

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在PyPI - the Python Package Index上注册，只要找到对应的模块名字，即可用easy_install或者pip安装。

本章介绍常用的第三方模块。

PIL

PIL：Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

安装PIL

在Debian/Ubuntu Linux下直接通过apt安装：

```
$ sudo apt-get install python-imaging
```

Mac和其他版本的Linux可以直接使用easy_install或pip安装，安装前需要把编译环境装好：

```
$ sudo easy_install PIL
```

如果安装失败，根据提示先把缺失的包（比如openjpeg）装上。

Windows平台就去PIL官方网站下载exe安装包。

操作图像

来看看最常见的图像缩放操作，只需三四行代码：

```
import Image

# 打开一个jpg图像文件，注意路径要改成你自己的:
im = Image.open('/Users/michael/test.jpg')
# 获得图像尺寸:
w, h = im.size
# 缩放到50%:
im.thumbnail((w//2, h//2))
# 把缩放后的图像用jpeg格式保存:
im.save('/Users/michael/thumbnail.jpg', 'jpeg')
```

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

```
import Image, ImageFilter

im = Image.open('/Users/michael/test.jpg')
im2 = im.filter(ImageFilter.BLUR)
im2.save('/Users/michael/blur.jpg', 'jpeg')
```

效果如下：



PIL的 `ImageDraw` 提供了一系列绘图方法，让我们可以直接绘图。比如要生成字母验证码图片：

```
import Image, ImageDraw, ImageFont, ImageFilter
import random

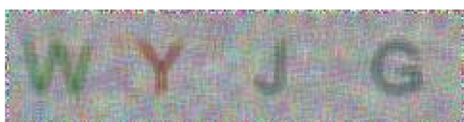
# 随机字母:
def rndChar():
    return chr(random.randint(65, 90))

# 随机颜色1:
def rndColor():
    return (random.randint(64, 255), random.randint(64, 255), rando

# 随机颜色2:
def rndColor2():
    return (random.randint(32, 127), random.randint(32, 127), rando

# 240 x 60:
width = 60 * 4
height = 60
image = Image.new('RGB', (width, height), (255, 255, 255))
# 创建Font对象:
font = ImageFont.truetype('Arial.ttf', 36)
# 创建Draw对象:
draw = ImageDraw.Draw(image)
# 填充每个像素:
for x in range(width):
    for y in range(height):
        draw.point((x, y), fill=rndColor())
# 输出文字:
for t in range(4):
    draw.text((60 * t + 10, 10), rndChar(), font=font, fill=rndColo
# 模糊:
image = image.filter(ImageFilter.BLUR)
image.save('code.jpg', 'jpeg');
```

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：



如果运行的时候报错：

```
IOError: cannot open resource
```

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

```
'/Library/Fonts/Arial.ttf'
```

要详细了解PIL的强大功能，请请参考PIL官方文档：

http://effbot.org/imagingbook/

图形界面

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets
- Qt
- GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

第一步是导入Tkinter包的所有内容：

```
from Tkinter import *
```

第二步是从 `Frame` 派生一个 `Application` 类，这是所有Widget的父容器：

```
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.helloLabel = Label(self, text='Hello, world!')
        self.helloLabel.pack()
        self.quitButton = Button(self, text='Quit', command=self.qu
        self.quitButton.pack()
```

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

`pack()` 方法把Widget加入到父容器中，并实现布局。 `pack()` 是最简单的布局， `grid()` 可以实现更复杂的布局。

在 `createWidgets()` 方法中，我们创建一个 `Label` 和一个 `Button` ，当Button被点击时，触发 `self.quit()` 使程序退出。

第三步，实例化 `Application` ，并启动消息循环：

```
app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：



点击“Quit”按钮或者窗口的“x”结束程序。

输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

```
from Tkinter import *
import tkMessageBox

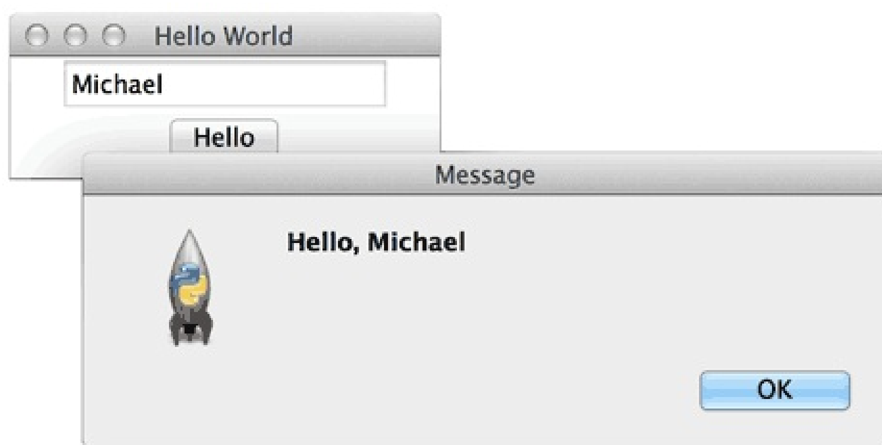
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.nameInput = Entry(self)
        self.nameInput.pack()
        self.alertButton = Button(self, text='Hello', command=self.
        self.alertButton.pack()

    def hello(self):
        name = self.nameInput.get() or 'world'
        tkMessageBox.showinfo('Message', 'Hello, %s' % name)
```

当用户点击按钮时，触发 `hello()`，通过 `self.nameInput.get()` 获得用户输入的文本后，使用 `tkMessageBox.showinfo()` 可以弹出消息对话框。

程序运行结果如下：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

源码参考：https://github.com/michaelliao/learn-python/tree/master/gui

网络编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。



网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

本章我们将详细介绍Python网络编程的概念和最主要的两种网络类型的编程。

TCP/IP简介

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

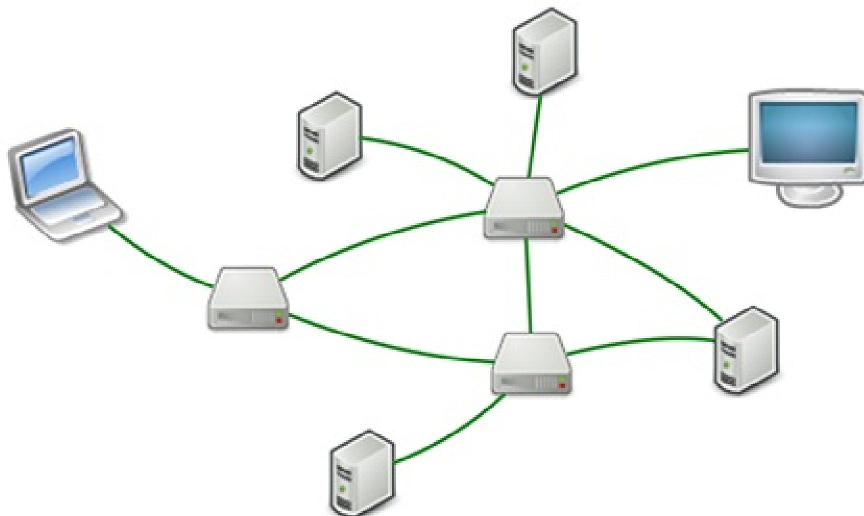
计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似123.123.123.123。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。



TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个IP包除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个网络程序。一个IP包来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。

TCP编程

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

客户端

大多数连接都是可靠的TCP连接。创建TCP连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

```
# 导入socket库:
import socket
# 创建一个socket:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('www.sina.com.cn', 80))
```

创建 `Socket` 时， `AF_INET` 指定使用IPv4协议，如果要用更先进的IPv6，就指定为 `AF_INET6` 。 `SOCK_STREAM` 指定使用面向流的TCP协议，这样，一个 `Socket` 对象就创建成功，但是还没有建立连接。

客户端要主动发起TCP连接，必须知道服务器的IP地址和端口号。新浪网站的IP地址可以用域名 `www.sina.com.cn` 自动转换到IP地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在 `80` 端口，因为 `80` 端口是Web服务的标准端口。其他服务都有对应的标准端口号，例如SMTP

服务是25端口，FTP服务是21端口，等等。端口号小于1024的是Internet标准服务的端口，端口号大于1024的，可以任意使用。

因此，我们连接新浪服务器的代码如下：

```
s.connect(('www.sina.com.cn', 80))
```

注意参数是一个tuple，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

```
# 发送数据:
s.send('GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: clos
```

TCP连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合HTTP标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

```
# 接收数据:
buffer = []
while True:
    # 每次最多接收1k字节:
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break
data = ''.join(buffer)
```

接收数据时，调用 `recv(max)` 方法，一次最多接收指定的字节数，因此，在一个while循环中反复接收，直到 `recv()` 返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用 `close()` 方法关闭Socket，这样，一次完整的网络通信就结束了：

```
# 关闭连接:
s.close()
```

接收到的数据包括HTTP头和网页本身，我们只需要把HTTP头和网页分离一下，把HTTP头打印出来，网页内容保存到文件：

```
header, html = data.split('\r\n\r\n', 1)
print header
# 把接收的数据写入文件:
with open('sina.html', 'wb') as f:
    f.write(html)
```

现在，只需要在浏览器中打开这个 `sina.html` 文件，就可以看到新浪的首页了。

服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立Socket连接，随后的通信就靠这个Socket连接了。

所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就创建该Socket连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个Socket连接是和哪个客户端绑定的。一个Socket依赖4项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个Socket。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上 `Hello` 再发回去。

首先，创建一个基于IPv4和TCP协议的Socket：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的IP地址上，也可以用 `0.0.0.0` 绑定到所有的网络地址，还可以用 `127.0.0.1` 绑定到本机地址。 `127.0.0.1` 是一个特殊的IP地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用 `9999` 这个端口号。请注意，小于 `1024` 的端口号必须要有管理员权限才能绑定：

```
# 监听端口:
s.bind(('127.0.0.1', 9999))
```

紧接着，调用 `listen()` 方法开始监听端口，传入的参数指定等待连接的最大数量：

```
s.listen(5)
print 'Waiting for connection...'
```

接下来，服务器程序通过一个永久循环来接受来自客户端的连接， `accept()` 会等待并返回一个客户端的连接:

```
while True:
    # 接受一个新连接:
    sock, addr = s.accept()
    # 创建新线程来处理TCP连接:
    t = threading.Thread(target=tcplink, args=(sock, addr))
    t.start()
```

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接：

```
def tcplink(sock, addr):
    print 'Accept new connection from %s:%s...' % addr
    sock.send('Welcome!')
    while True:
        data = sock.recv(1024)
        time.sleep(1)
        if data == 'exit' or not data:
            break
        sock.send('Hello, %s!' % data)
    sock.close()
    print 'Connection from %s:%s closed.' % addr
```

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上 `Hello` 再发送给客户端。如果客户端发送了 `exit` 字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('127.0.0.1', 9999))
# 接收欢迎消息:
print s.recv(1024)
for data in ['Michael', 'Tracy', 'Sarah']:
    # 发送数据:
    s.send(data)
    print s.recv(1024)
s.send('exit')
s.close()
```

我们需要打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：

```
socket — Python — 50×11
Mavericks:socket michael$ python echo_server.py
Waiting for connection...
Accept new connection from 127.0.0.1:64398...
Connection from 127.0.0.1:64398 closed.
```

```
socket — bash — 50×11
Mavericks:socket michael$ python echo_client.py
Welcome!
Hello, Michael!
Hello, Tracy!
Hello, Sarah!
Mavericks:socket michael$
```

需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

小结

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

源码参考：https://github.com/michaelliao/learn-python/tree/master/socket

UDP编程

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务器。服务器首先需要绑定端口：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 绑定端口:
s.bind(('127.0.0.1', 9999))
```

创建Socket时， `SOCK_DGRAM` 指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用 `listen()` 方法，而是直接接收来自任何客户端的数据：

```
print 'Bind UDP on 9999...'
while True:
    # 接收数据:
    data, addr = s.recvfrom(1024)
    print 'Received from %s:%s.' % addr
    s.sendto('Hello, %s!' % data, addr)
```

`recvfrom()` 方法返回数据和客户端的地址与端口，这样，服务器收到数据后，直接调用 `sendto()` 就可以把数据用UDP发给客户端。

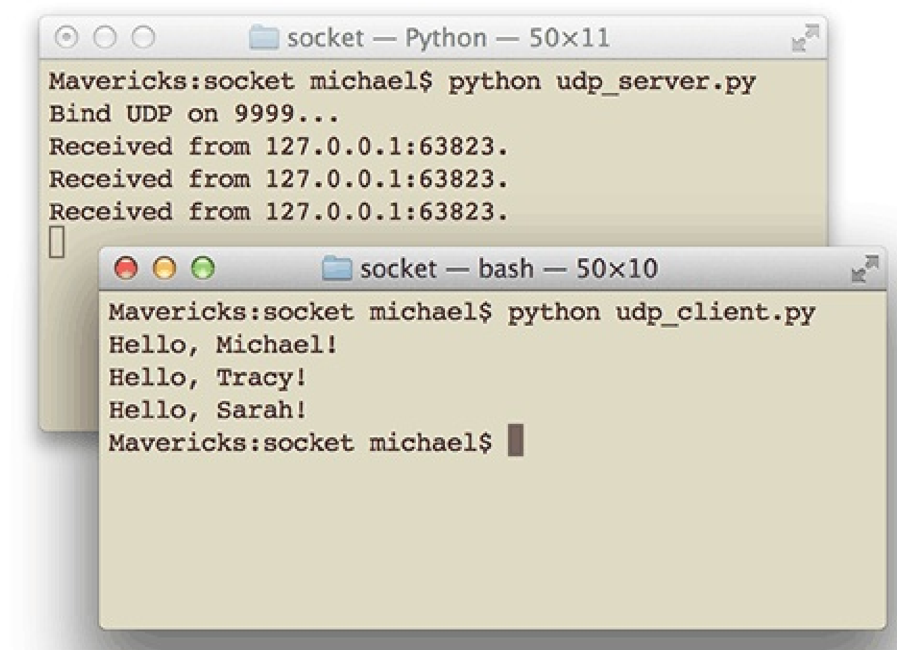
注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用 `connect()` ，直接通过 `sendto()` 给服务器发数据：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for data in ['Michael', 'Tracy', 'Sarah']:
    # 发送数据:
    s.sendto(data, ('127.0.0.1', 9999))
    # 接收数据:
    print s.recv(1024)
s.close()
```

从服务器接收数据仍然调用 `recv()` 方法。

仍然用两个命令行分别启动服务器和客户端测试，结果如下：



小结

UDP的使用与TCP类似，但是不需要建立连接。此外，服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

源码参考：https://github.com/michaelliao/learn-python/tree/master/socket

电子邮件

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发到天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是 `me@163.com` ，对方的电子邮件地址是 `friend@sina.com` （注意地址都是虚构的哈），现在我们用 `Outlook` 或者 `Foxmail` 之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为**MUA**：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到**MTA**：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是 `163.com` ，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是 `@sina.com` 的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地**MDA**：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

```
发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人
```

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

发邮件时，MUA和MTA使用的协议就是SMTP：Simple Mail Transfer Protocol，后面的MTA到另一个MTA也是用SMTP协议。

收邮件时，MUA和MDA使用的协议有两种：POP：Post Office Protocol，目前版本是3，俗称POP3；IMAP：Internet Message Access Protocol，目前版本是4，优点是不但能取邮件，还可以直接操作MDA上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置SMTP服务器，也就是你要发到哪个MTA上。假设你正在使用163的邮箱，你就不能直接发到新浪的MTA上，因为它只服务新浪的用户，所以，你得填163提供的SMTP服务器地址： `smtp.163.com` ，为了证明你是163的用户，SMTP服务器还要求你填写邮箱地址和邮箱口令，这样，MUA才能正常地把Email通过SMTP协议发送到MTA。

类似的，从MDA收邮件时，MDA服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook之类的邮件客户端会要求你填写POP3或IMAP服务器地址、邮箱地址和口令，这样，MUA才能顺利地通过POP或IMAP协议从MDA取到邮件。

在使用Python收发邮件前，请先准备好至少两个电子邮件，如 `xxx@163.com` ， `xxx@sina.com` ， `xxx@qq.com` 等，注意两个邮箱不要用同一家邮件服务商。

SMTP发送邮件

SMTP是发送邮件的协议，Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。

Python对SMTP支持有 `smtplib` 和 `email` 两个模块， `email` 负责构造邮件， `smtplib` 负责发送邮件。

首先，我们来构造一个最简单的纯文本邮件：

```
from email.mime.text import MIMEText
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
```

注意到构造 `MIMEText` 对象时，第一个参数就是邮件正文，第二个参数是MIME的subtype，传入 `'plain'` ，最终的MIME就是 `'text/plain'` ，最后一定要用 `utf-8` 编码保证多语言兼容性。

然后，通过SMTP发出去：

```
# 输入Email地址和口令:
from_addr = raw_input('From: ')
password = raw_input('Password: ')
# 输入SMTP服务器地址:
smtp_server = raw_input('SMTP server: ')
# 输入收件人地址:
to_addr = raw_input('To: ')

import smtplib
server = smtplib.SMTP(smtp_server, 25) # SMTP协议默认端口是25
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们用 `set_debuglevel(1)` 就可以打印出和SMTP服务器交互的所有信息。SMTP协议就是简单的文本命令和响应。 `login()` 方法用来登录SMTP服务器， `sendmail()` 方法就是发邮件，由于可以一次发给多个人，所以传入一

个 `list` ，邮件正文是一个 `str` ， `as_string()` 把 `MIMEText` 对象变成 `str` 。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：



hello, send by Python...

仔细观察，发现如下问题：

1. 邮件没有主题；
2. 收件人的名字没有显示为友好的名字，比如 `Mr Green <green@example.com>` ；
3. 明明收到了邮件，却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过SMTP协议发给MTA，而是包含在发给MTA的文本中的，所以，我们必须把 `From` 、 `To` 和 `Subject` 添加到 `MIMEText` 中，才是一封完整的邮件：

```
# -*- coding: utf-8 -*-

from email import encoders
from email.header import Header
from email.mime.text import MIMEText
from email.utils import parseaddr, formataddr
import smtplib

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr(( \
        Header(name, 'utf-8').encode(), \
        addr.encode('utf-8') if isinstance(addr, unicode) else addr

from_addr = raw_input('From: ')
password = raw_input('Password: ')
to_addr = raw_input('To: ')
smtp_server = raw_input('SMTP server: ')

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
msg['From'] = _format_addr(u'Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr(u'管理员 <%s>' % to_addr)
msg['Subject'] = Header(u'来自SMTP的问候……', 'utf-8').encode()

server = smtplib.SMTP(smtp_server, 25)
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们编写了一个函数 `_format_addr()` 来格式化一个邮件地址。注意不能简单地传入 `name <addr@example.com>` ，因为如果包含中文，需要通过 `Header` 对象进行编码。

`msg['To']` 接收的是字符串而不是list，如果有多个邮件地址，用 `,` 分隔即可。

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：

来自SMTP的问候……

发件人：Python爱好者 <xxxxxx@163.com>

时　间：2014年8月14日(星期四) 下午3:45

收件人：管理员 <xxxxxx@qq.com>

hello, send by Python...

你看到的收件人的名字很可能不是我们传入的 `管理员` ，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

```
From: =?utf-8?b?UHl0aG9u54ix5aW96ICF?= <xxxxxx@163.com>
To: =?utf-8?b?566h55CG5ZGY?= <xxxxxx@qq.com>
Subject: =?utf-8?b?5p2l6IeqU01UUOeahOmXruWAmeKApuKApg==?=
```

这就是经过 `Header` 对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。

发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造 `MIMEText` 对象时，把HTML字符串传进去，再把第二个参数由 `plain` 变为 `html` 就可以了：

```
msg = MIMEText('<html><body><h1>Hello</h1>' +
    '<p>send by <a href="http://www.python.org">Python</a>...</p>'
    '</body></html>', 'html', 'utf-8')
```

再发送一遍邮件，你将看到以HTML显示的邮件：

来自SMTP的问候......

发件人：Python爱好者 <xxxxxx@163.com>

时　间：2014年8月14日(星期四) 下午4:06

收件人：管理员 <xxxxxx@qq.com>

Hello

send by Python...

发送附件

如果Email中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个 `MIMEMultipart` 对象代表邮件本身，然后往里面加上一个 `MIMEText` 作为邮件正文，再继续往里面加上表示附件的 `MIMEBase` 对象即可：

```
# 邮件对象:
msg = MIMEMultipart()
msg['From'] = _format_addr(u'Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr(u'管理员 <%s>' % to_addr)
msg['Subject'] = Header(u'来自SMTP的问候……', 'utf-8').encode()

# 邮件正文是MIMEText:
msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

# 添加附件就是加上一个MIMEBase，从本地读取一个图片:
with open('/Users/michael/Downloads/test.png', 'rb') as f:
    # 设置附件的MIME和文件名，这里是png类型:
    mime = MIMEBase('image', 'png', filename='test.png')
    # 加上必要的头信息:
    mime.add_header('Content-Disposition', 'attachment', filename=
    mime.add_header('Content-ID', '<0>')
    mime.add_header('X-Attachment-Id', '0')
    # 把附件的内容读进来:
    mime.set_payload(f.read())
    # 用Base64编码:
    encoders.encode_base64(mime)
    # 添加到MIMEMultipart:
    msg.attach(mime)
```

然后，按正常发送流程把 `msg` （注意类型已变为 `MIMEMultipart` ）发送出去，就可以收到如下带附件的邮件：

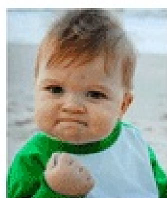
来自SMTP的问候……

发件人：Python爱好者 <xxxxxx@163.com>
时　间：2014年8月14日(星期四) 下午5:08
收件人：管理员 <xxxxxx@qq.com>
附　件：1 个 (test.png)

send with file...

附件(1 个)

普通附件



test.png (80.13K)
下载 预览 收藏 转存

发送图片

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。

要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用 `src="cid:0"` 就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的 `cid:x` 即可。

把上面代码加入 `MIMEMultipart` 的 `MIMEText` 从 `plain` 改为 `html` ，然后在适当的位置引用图片：

```
msg.attach(MIMEText('<html><body><h1>Hello</h1>' +
    '<p><img src="cid:0"></p>' +
    '</body></html>', 'html', 'utf-8'))
```

再次发送，就可以看到图片直接嵌入到邮件正文的效果：

来自SMTP的问候……

发件人：Python爱好者 <asklxf@163.com>

时　间：2014年8月14日(星期四) 下午5:27

收件人：Xuefeng <18224514@qq.com>

Hello



同时支持HTML和Plain格式

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用 `MIMEMultipart` 就可以组合一个HTML和Plain，要注意指定subtype是 `alternative` ：

```
msg = MIMEMultipart('alternative')
msg['From'] = ...
msg['To'] = ...
msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))
msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'ht
# 正常发送msg对象...
```

加密SMTP

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：

```
smtp_server = 'smtp.gmail.com'
smtp_port = 587
server = smtplib.SMTP(smtp_server, smtp_port)
server.starttls()
# 剩下的代码和前面的一模一样:
server.set_debuglevel(1)
...
```

只需要在创建 `SMTP` 对象后，立刻调用 `starttls()` 方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

小结

使用Python的smtplib发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个 `Messag` 对象，如果构造一个 `MIMEText` 对象，就表示一个文本邮件对象，如果构造一个 `MIMEImage` 对象，就表示一个作为附件的图片，要把多个对象组合起来，就用 `MIMEMultipart` 对象，而 `MIMEBase` 可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
   +- MIMEMultipart
   +- MIMENonMultipart
      +- MIMEMessage
      +- MIMEText
      +- MIMEImage
```

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过email.mime文档查看它们所在的包以及详细的用法。

源码参考：

https://github.com/michaelliao/learn-python/tree/master/email

POP3收取邮件

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个**MUA**作为客户端，从**MDA**把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是**POP**协议，目前版本号是3，俗称**POP3**。

Python内置一个 `poplib` 模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用 `email` 模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用 `poplib` 把邮件的原始文本下载到本地；

第二部：用 `email` 解析原始文本，还原为邮件对象。

通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

```
import poplib

# 输入邮件地址, 口令和POP3服务器地址:
email = raw_input('Email: ')
password = raw_input('Password: ')
pop3_server = raw_input('POP3 server: ')

# 连接到POP3服务器:
server = poplib.POP3(pop3_server)
# 可以打开或关闭调试信息:
# server.set_debuglevel(1)
# 可选:打印POP3服务器的欢迎文字:
print(server.getwelcome())
# 身份认证:
server.user(email)
server.pass_(password)
# stat()返回邮件数量和占用空间:
print('Messages: %s. Size: %s' % server.stat())
# list()返回所有邮件的编号:
resp, mails, octets = server.list()
# 可以查看返回的列表类似['1 82923', '2 2184', ...]
print(mails)
# 获取最新一封邮件, 注意索引号从1开始:
index = len(mails)
resp, lines, octets = server.retr(index)
# lines存储了邮件的原始文本的每一行,
# 可以获得整个邮件的原始文本:
msg_content = '\r\n'.join(lines)
# 稍后解析出邮件:
msg = Parser().parsestr(msg_content)
# 可以根据邮件索引号直接从服务器删除邮件:
# server.dele(index)
# 关闭连接:
server.quit()
```

用POP3获取邮件其实很简单，要获取所有邮件，只需要循环使用 `retr()` 把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

解析邮件

解析邮件的过程和上一节构造邮件正好相反，因此，先导入必要的模块：

```
import email
from email.parser import Parser
from email.header import decode_header
from email.utils import parseaddr
```

只需要一行代码就可以把邮件内容解析为 `Message` 对象：

```
msg = Parser().parsestr(msg_content)
```

但是这个 `Message` 对象本身可能是一个 `MIMEMultipart` 对象，即包含嵌套的其他 `MIMEBase` 对象，嵌套可能还不止一层。

所以我们要递归地打印出 `Message` 对象的层次结构：

```
# indent用于缩进显示:
def print_info(msg, indent=0):
    if indent == 0:
        # 邮件的From, To, Subject存在于根对象上:
        for header in ['From', 'To', 'Subject']:
            value = msg.get(header, '')
            if value:
                if header=='Subject':
                    # 需要解码Subject字符串:
                    value = decode_str(value)
                else:
                    # 需要解码Email地址:
                    hdr, addr = parseaddr(value)
                    name = decode_str(hdr)
                    value = u'%s <%s>' % (name, addr)
            print('%s%s: %s' % ('  ' * indent, header, value))
    if (msg.is_multipart()):
        # 如果邮件对象是一个MIMEMultipart,
        # get_payload()返回list，包含所有的子对象:
        parts = msg.get_payload()
```

```
        for n, part in enumerate(parts):
            print('%spart %s' % ('  ' * indent, n))
            print('%s--------------------' % ('  ' * indent))
            # 递归打印每一个子对象:
            print_info(part, indent + 1)
    else:
        # 邮件对象不是一个MIMEMultipart,
        # 就根据content_type判断:
        content_type = msg.get_content_type()
        if content_type=='text/plain' or content_type=='text/html'
            # 纯文本或HTML内容:
            content = msg.get_payload(decode=True)
            # 要检测文本编码:
            charset = guess_charset(msg)
            if charset:
                content = content.decode(charset)
            print('%sText: %s' % ('  ' * indent, content + '...'))
        else:
            # 不是文本,作为附件处理:
            print('%sAttachment: %s' % ('  ' * indent, content_type
```

邮件的Subject或者Email中包含的名字都是经过编码后的str，要正常显示，就必须decode：

```
def decode_str(s):
    value, charset = decode_header(s)[0]
    if charset:
        value = value.decode(charset)
    return value
```

`decode_header()` 返回一个list，因为像 `Cc` 、 `Bcc` 这样的字段可能包含多个邮件地址，所以解析出来的会有多个元素。上面的代码我们偷了个懒，只取了第一个元素。

文本邮件的内容也是str，还需要检测编码，否则，非UTF-8编码的邮件都无法正常显示：

```
def guess_charset(msg):
    # 先从msg对象获取编码:
    charset = msg.get_charset()
    if charset is None:
        # 如果获取不到，再从Content-Type字段获取:
        content_type = msg.get('Content-Type', '').lower()
        pos = content_type.find('charset=')
        if pos >= 0:
            charset = content_type[pos + 8:].strip()
    return charset
```

把上面的代码整理好，我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件，然后用浏览器登录邮箱，看看邮件收到没，如果收到了，我们就来用Python程序把它收到本地：



Python可以使用POP3收取邮件......

运行程序，结果如下：

```
+OK Welcome to coremail Mail Pop3 Server (163coms[...])
Messages: 126\. Size: 27228317

From: Test <xxxxxx@qq.com>
To: Python爱好者 <xxxxxx@163.com>
Subject: 用POP3收取邮件
part 0
--------------------
  part 0
  --------------------
    Text: Python可以使用POP3收取邮件……...
  part 1
  --------------------
    Text: Python可以<a href="...">使用POP3</a>收取邮件……...
part 1
--------------------
  Attachment: application/octet-stream
```

我们从打印的结构可以看出，这封邮件是一个 `MIMEMultipart` ，它包含两部分：第一部分又是一个 `MIMEMultipart` ，第二部分是一个附件。而内嵌的 `MIMEMultipart` 是一个 `alternative` 类型，它包含一个纯文本格式的 `MIMEText` 和一个HTML格式的 `MIMEText` 。

小结

用Python的 `poplib` 模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用 `email` 模块把原始邮件解析为 `Message` 对象，然后，用适当的形式把邮件内容展示给用户即可。

源码参考：

https://github.com/michaelliao/learn-python/tree/master/email

访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用 `,` 隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name":"Michael","score":99},
    {"name":"Bob","score":85},
    {"name":"Bart","score":59},
    {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

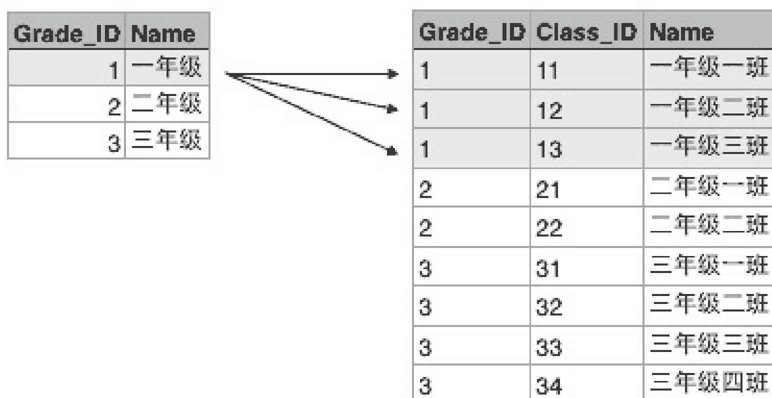
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



Grade_ID	Name
1	一年级
2	二年级
3	三年级

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

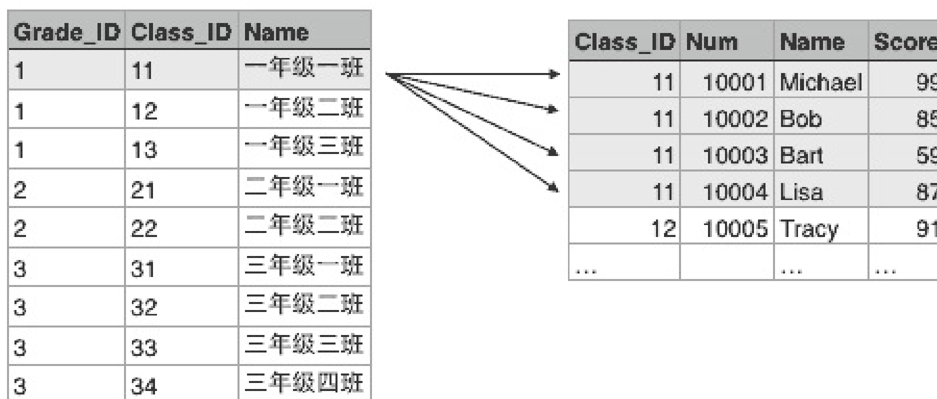
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
---------+----------+----------
grade_id | class_id | name
---------+----------+----------
1        | 11       | 一年级一班
---------+----------+----------
1        | 12       | 一年级二班
---------+----------+----------
1        | 13       | 一年级三班
---------+----------+----------
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...		...	...

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，推荐Coursera课程：

英文：https://www.coursera.org/course/db

中文：http://c.open.163.com/coursera/courseIntro.htm?cid=12

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无

一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装MySQL Community Server 5.6，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

```
# 导入SQLite驱动:
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建:
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor:
>>> cursor = conn.cursor()
# 执行一条SQL语句，创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key,
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录:
>>> cursor.execute('insert into user (id, name) values (\'1\', \'Mi
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数:
>>> cursor.rowcount
1
# 关闭Cursor:
>>> cursor.close()
# 提交事务:
>>> conn.commit()
# 关闭Connection:
>>> conn.close()
```

我们再试试查询记录：

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句:
>>> cursor.execute('select * from user where id=?', '1')
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集:
>>> values = cursor.fetchall()
>>> values
[(u'1', u'Michael')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时，只要搞清楚Connection和Cursor对象，打开后一定记得关闭，就可以放心地使用。

使用Cursor对象执行 `insert` ， `update` ， `delete` 语句时，执行结果由 `rowcount` 返回影响的行数，就可以拿到执行结果。

使用Cursor对象执行 `select` 语句时，通过 `featchall()` 可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给 `execute()` 方法，有几个 `?` 占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where id=?', '1')
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过Connection对象和Cursor对象操作数据。

要确保打开的Connection对象和Cursor对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉Connection对象和Cursor对象呢？请回忆 `try:...except:...finally:...` 的用法。

使用MySQL

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的Community Server 5.6.x版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password` 。

在Windows上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf` ：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+--------------------------+----------------------------------------
| Variable_name            | Value
+--------------------------+----------------------------------------
| character_set_client     | utf8
| character_set_connection | utf8
| character_set_database   | utf8
| character_set_filesystem | binary
| character_set_results    | utf8
| character_set_server     | utf8
| character_set_system     | utf8
| character_sets_dir       | /usr/local/mysql-5.1.65-osx10.6-x86_64
+--------------------------+----------------------------------------
8 rows in set (0.00 sec)
```

看到 `utf8` 字样就表示编码设置正确。

安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。

目前，有两个MySQL驱动：

- mysql-connector-python：是MySQL官方的纯Python驱动；
- MySQL-python：是封装了MySQL C驱动的Python驱动。

可以把两个都装上，使用的时候再决定用哪个：

```
$ easy_install mysql-connector-python
$ easy_install MySQL-python
```

我们以mysql-connector-python为例，演示如何连接到MySQL服务器的test数据库：

```
# 导入MySQL驱动:
>>> import mysql.connector
# 注意把password设为你的root口令:
>>> conn = mysql.connector.connect(user='root', password='password
>>> cursor = conn.cursor()
# 创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key,
# 插入一行记录，注意MySQL的占位符是%s:
>>> cursor.execute('insert into user (id, name) values (%s, %s)', [
>>> cursor.rowcount
1
# 提交事务:
>>> conn.commit()
>>> cursor.close()
# 运行查询:
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s', '1')
>>> values = cursor.fetchall()
>>> values
[(u'1', u'Michael')]
# 关闭Cursor和Connection:
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

小结

- MySQL的SQL占位符是 `%s` ；
- 通常我们在连接MySQL时传入 `use_unicode=True` ，让MySQL的DB-API始终返回Unicode。

使用SQLAlchemy

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含 `id` 和 `name` 的 `user` 表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过easy_install或者pip安装SQLAlchemy：

```
$ easy_install sqlalchemy
```

然后，利用上次我们在MySQL的test数据库中创建的 `user` 表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

```
# 导入:
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# 创建对象的基类:
Base = declarative_base()

# 定义User对象:
class User(Base):
    # 表的名字:
    __tablename__ = 'user'

    # 表的结构:
    id = Column(String(20), primary_key=True)
    name = Column(String(20))

# 初始化数据库连接:
engine = create_engine('mysql+mysqlconnector://root:password@localh
# 创建DBSession类型:
DBSession = sessionmaker(bind=engine)
```

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

```
class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...
```

`create_engine()` 用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

```
'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
```

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个 `User` 对象：

```
# 创建session对象:
session = DBSession()
# 创建新User对象:
new_user = User(id='5', name='Bob')
# 添加到session:
session.add(new_user)
# 提交即保存到数据库:
session.commit()
# 关闭session:
session.close()
```

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。Session对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是 `User` 对象。SQLAlchemy提供的查询接口如下：

```
# 创建Session:
session = DBSession()
# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的name属性:
print 'type:', type(user)
print 'name:', user.name
# 关闭Session:
session.close()
```

运行结果如下：

```
type: <class '__main__.User'>
name: Bob
```

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')

class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # “多”的一方的book表是通过外键关联到user表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

Web开发

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

1. 静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；
2. CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。
3. ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。

4. MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

Python的诞生历史比Web还要早，由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。

Python有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

本章我们会详细讨论Python Web开发技术。

HTTP协议简介

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器之间的传输协议是HTTP，所以：

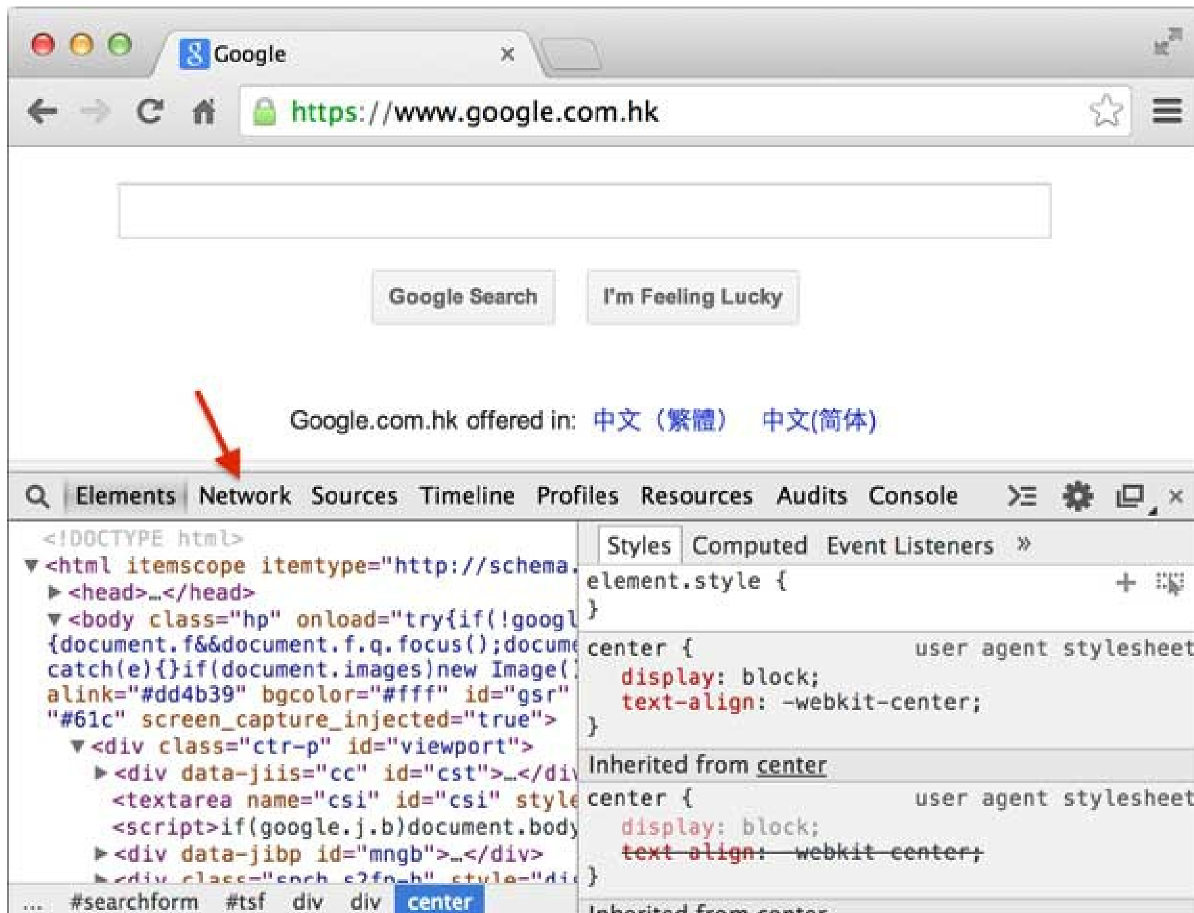
- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

在举例子之前，我们需要安装Google的Chrome浏览器。

为什么要使用Chrome浏览器而不是IE呢？因为IE实在是太慢了，并且，IE对于开发和调试Web应用程序完全是一点用也没有。

我们需要在浏览器很方便地调试我们的Web应用，而Chrome提供了一套完整地调试工具，非常适合Web开发。

安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



`Elements` 显示网页的结构， `Network` 显示浏览器和服务器的通信。我们点 `Network` ，确保第一个小红灯亮着，Chrome就会记录所有浏览器和服务器之间的通信：



当我们在地址栏输入 `www.sina.com.cn` 时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过 `Network` 的记录，我们就可以知道。在 `Network` 中，定位到第一条记录，点击，右侧将显示 `Request Headers` ，点击右侧的 `view source` ，我们就可以看到浏览器发给新浪服务器的请求：

最主要的头两行分析如下，第一行：

```
GET / HTTP/1.1
```

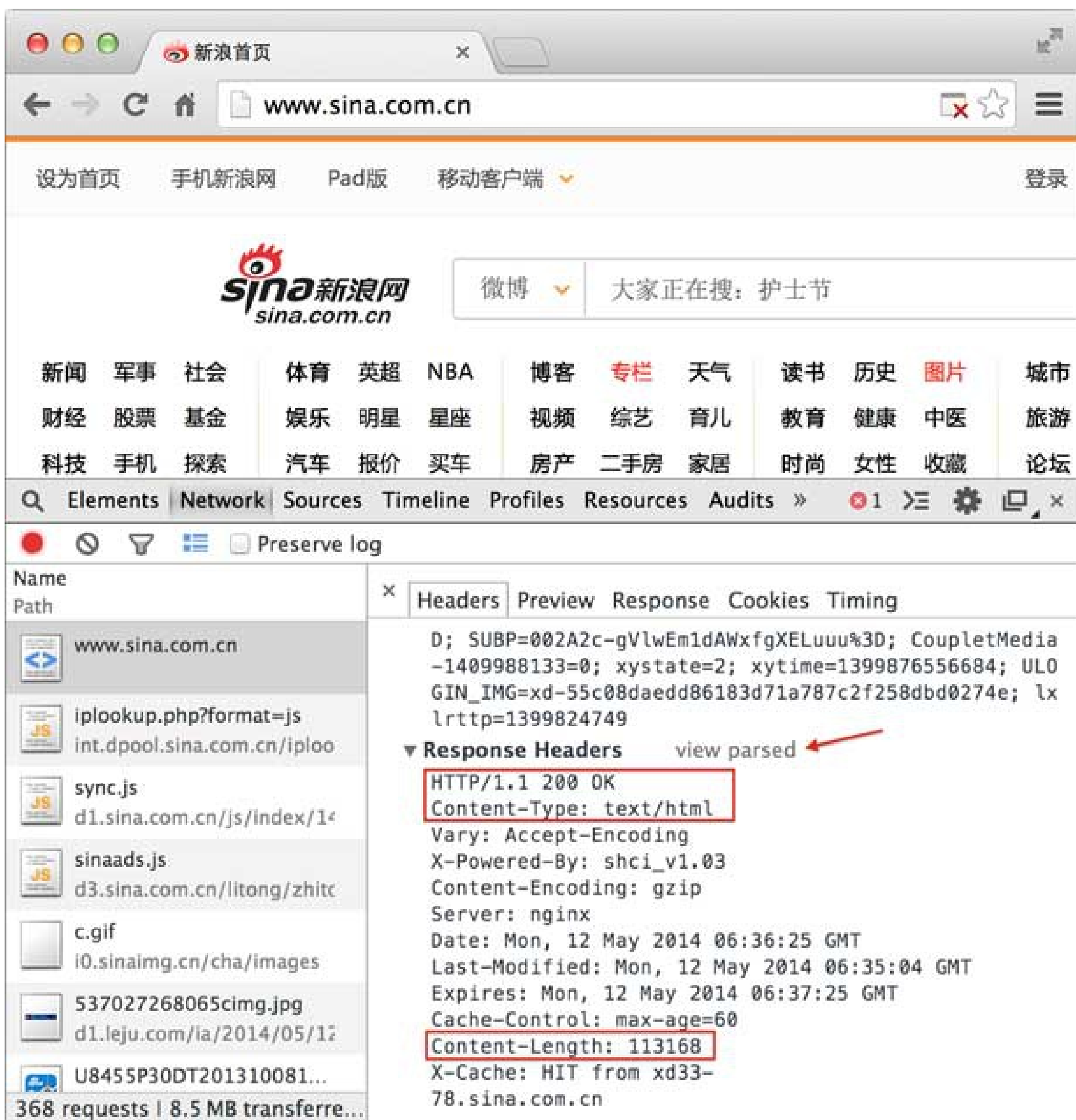
`GET` 表示一个读取请求，将从服务器获得网页数据， `/` 表示URL的路径，URL总是以 `/` 开头， `/` 就表示首页，最后的 `HTTP/1.1` 指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用一个TCP连接，以加快传输速度。

从第二行开始，每一行都类似于 `Xxx: abcdefg` ：

```
Host: www.sina.com.cn
```

表示请求的域名是 `www.sina.com.cn` 。如果一台服务器有多个网站，服务器就需要通过 `Host` 来区分浏览器请求的是哪个网站。

继续往下找到 `Response Headers` ，点击 `view source` ，显示服务器返回的原始响应数据：



HTTP响应分为Header和Body两部分（Body是可选项），我们在 `Network` 中看到的Header最重要的几行如下：

```
200 OK
```

`200` 表示一个成功的响应，后面的 `OK` 是说明。失败的响应有 `404 Not Found` ：网页不存在， `500 Internal Server Error` ：服务器内部出错，等等。

```
Content-Type: text/html
```

`Content-Type` 指示响应的内容，这里是 `text/html` 表示HTML网页。请注意，浏览器就是依靠 `Content-Type` 来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠URL来判断响应的内容，所以，即使URL是 `http://example.com/abc.jpg` ，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：

```
新浪首页    view-source:www.sina.com.cn
view-source:www.sina.com.cn

1  <!DOCTYPE html>
2  <!--[30,131,1] published at 2014-05-12 14:35:02 from #153 by 9018-->
3  <html>
4  <head>
5      <meta http-equiv="Content-type" content="text/html; charset=gb2312"
   />
6      <title>新浪首页</title>
7
8          <meta name="keywords" content="新浪,新浪网,SINA,sina,sina.com.cn,
   新浪首页,门户,资讯" />
9          <meta name="description" content="新浪网为全球用户24小时提供全面及时
   的中文资讯，内容覆盖国内外突发新闻事件、体坛赛事、娱乐时尚、产业资讯、实用信息等，设有
   新闻、体育、娱乐、财经、科技、房产、汽车等30多个内容频道，同时开设博客、视频、论坛等自
   由互动交流空间。" />
10         <meta name="stencil" content="PGLS000022" />
11         <meta name="publishid" content="30,131,1" />
12         <meta name="verify-v1"
   content="6HtwmypggdgP1NLw7NOuQBI2TW8+CfkYCoyeB8IDbn8=" />
13         <meta name="360-site-verification"
   content="63349a2167ca11f4b9bd9a8d48354541" />
14         <meta name="application-name" content="新浪首页"/>
15         <meta name ="msapplication-TileImage"
   content="http://i1.sinaimg.cn/dy/deco/2013/0312/logo.png"/>
16         <meta name="msapplication-TileColor" content="#ffbf27"/>
17         <meta name="sogou_site_verification" content="BVIdHxKGrl"/>
18 <link rel="apple-touch-icon"
   href="http://i3.sinaimg.cn/home/2013/0331/U586P30DT20130331093840.png"
   />
19
20         <script type="text/javascript">
21     //js异步加载管理
22     (function(){var w=this,d=document,version='1.0.7',data=
   {},length=0,cbkLen=0;if(w.jsLoader){if(w.jsLoader.version>=version)
   {return};data=w.jsLoader.getData();length=data.length};var
   addEvent=function(obj,eventType,func){if(obj.attachEvent)
   {obj.attachEvent("on"+eventType,func)}else{obj.addEventListener(eventTy
   pe,func,false)}};var domReady=false,ondomReady=function()
```

当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在 `Network` 下面能看到很多额外的HTTP请求。

HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: www.sina.com.cn

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据。

步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是 `http://www.sina.com.cn/` 的首页，但是新浪在HTML中可以链入其他服务器的资源，比如 `<img src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">` ，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。HTTP GET请求的格式：

```
GET /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

每个Header一行一个，换行符是 `\r\n` 。

HTTP POST请求的格式：

```
POST /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

当遇到连续两个 `\r\n` 时，Header部分结束，后面的数据全部是Body。

HTTP响应的格式：

```
200 OK
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

HTTP响应如果包含body，也是通过 `\r\n\r\n` 来分隔的。请再次注意，Body的数据类型由 `Content-Type` 头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在 `Content-Encoding` 时，Body数据是被压缩的，最常见的压缩方式是gzip，所以，看到 `Content-Encoding: gzip` 时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

要详细了解HTTP协议，推荐“HTTP: The Definitive Guide”一书，非常不错，有中文译本：

HTTP权威指南

HTML简介

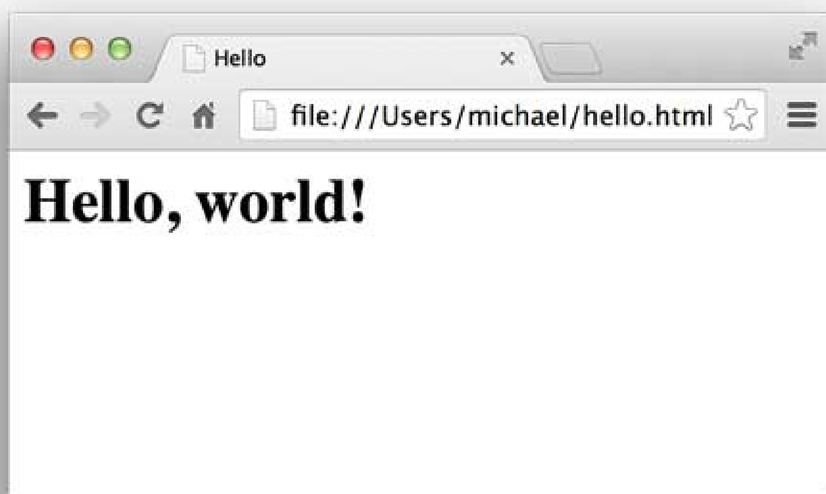
网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

HTML长什么样？上次我们看了新浪首页的HTML源码，如果仔细数数，竟然有6000多行！

所以，学HTML，就不要指望从新浪入手了。我们来看看最简单的HTML长什么样：

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

可以用文本编辑器编写HTML，然后保存为 `hello.html` ，双击或者把文件拖到浏览器中，就可以看到效果：



HTML文档就是一系列的Tag组成，最外层的Tag是 `<html>` 。规范的HTML也包含 `<head>...</head>` 和 `<body>...</body>` （注意不要和HTTP的Header、Body搞混了），由于HTML是富文档模型，所以，还有一系列的Tag用来表示链接、图片、表格、表单等等。

CSS简介

CSS是Cascading Style Sheets（层叠样式表）的简称，CSS用来控制HTML里的所有元素如何展现，比如，给标题元素 `<h1>` 加一个样式，变成48号字体，灰色，带阴影：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

效果如下：



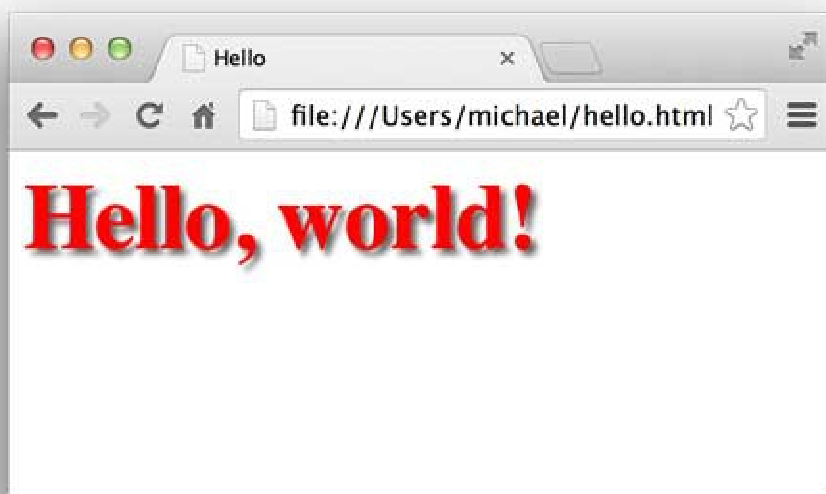
JavaScript简介

JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就

必须通过JavaScript来实现：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
  <script>
    function change() {
      document.getElementsByTagName('h1')[0].style.color = '#ff000
    }
  </script>
</head>
<body>
  <h1 onclick="change()">Hello, world!</h1>
</body>
</html>
```

效果如下：



小结

如果要学习Web开发，首先要对HTML、CSS和JavaScript作一定的了解。HTML定义了页面的内容，CSS来控制页面元素的样式，而JavaScript负责页面的交互逻辑。

讲解HTML、CSS和JavaScript就可以写3本书，对于优秀的Web开发人员来说，精通HTML、CSS和JavaScript是必须的，这里推荐一个在线学习网站w3schools：

http://www.w3schools.com/

以及一个对应的中文版本：

http://www.w3school.com.cn/

当我们用Python或者其他语言开发Web应用时，我们就是要在服务器端动态创建出HTML，这样，浏览器就会向不同的用户显示出不同的Web页面。

WSGI接口

了解了HTTP协议和HTML文档，我们其实就明白了一个Web应用的本质就是：

1. 浏览器发送一个HTTP请求；
2. 服务器收到请求，生成一个HTML文档；
3. 服务器把HTML文档作为HTTP响应的Body发送给浏览器；
4. 浏览器收到HTTP响应，从HTTP Body取出HTML文档并显示。

所以，最简单的Web应用就是先把HTML用文件保存好，用一个现成的HTTP服务器软件，接收用户请求，从文件中读取HTML，返回。Apache、Nginx、Lighttpd等这些常见的静态服务器就是干这件事情的。

如果要动态生成HTML，就需要把上述步骤自己来实现。不过，接受HTTP请求、解析HTTP请求、发送HTTP响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态HTML呢，就得花个把月去读HTTP规范。

正确的做法是底层代码由专门的服务器软件实现，我们用Python专注于生成HTML文档。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口，让我们专心用Python编写Web业务。

这个接口就是WSGI：Web Server Gateway Interface。

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello, web!”：

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, web!</h1>'
```

上面的`application()`函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

- environ：一个包含所有HTTP请求信息的`dict`对象；
- start_response：一个发送HTTP响应的函数。

在 `application()` 函数中，调用：

```
start_response('200 OK', [('Content-Type', 'text/html')])
```

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次 `start_response()` 函数。 `start_response()` 函数接收两个参数，一个是HTTP响应码，一个是一组 `list` 表示的HTTP Header，每个Header用一个包含两个 `str` 的 `tuple` 表示。

通常情况下，都应该把 `Content-Type` 头发送给浏览器。其他很多常用的HTTP Header也应该发送。

然后，函数的返回值 `'<h1>Hello, web!</h1>'` 将作为HTTP响应的Body发送给浏览器。

有了WSGI，我们关心的就是如何从 `environ` 这个 `dict` 对象拿到HTTP请求信息，然后构造HTML，通过 `start_response()` 发送Header，最后返回Body。

整个 `application()` 函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。

不过，等等，这个 `application()` 函数怎么调用？如果我们自己调用，两个参数 `environ` 和 `start_response` 我们没法提供，返回的 `str` 也没法发给浏览器。

所以 `application()` 函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器，我们可以挑选一个来用。但是现在，我们只想尽快测试一下我们编写的 `application()` 函数真的可以把HTML输出到浏览器，所以，要赶紧找一个最简单的WSGI服务器，把我们的Web应用程序跑起来。

好消息是Python内置了一个WSGI服务器，这个模块叫wsgiref，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

运行WSGI服务

我们先编写 `hello.py` ，实现Web应用程序的WSGI处理函数：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, web!</h1>'
```

然后，再编写一个 `server.py` ，负责启动WSGI服务器，加载 `application()` 函数：

```
# server.py
# 从wsgiref模块导入:
from wsgiref.simple_server import make_server
# 导入我们自己编写的application函数:
from hello import application

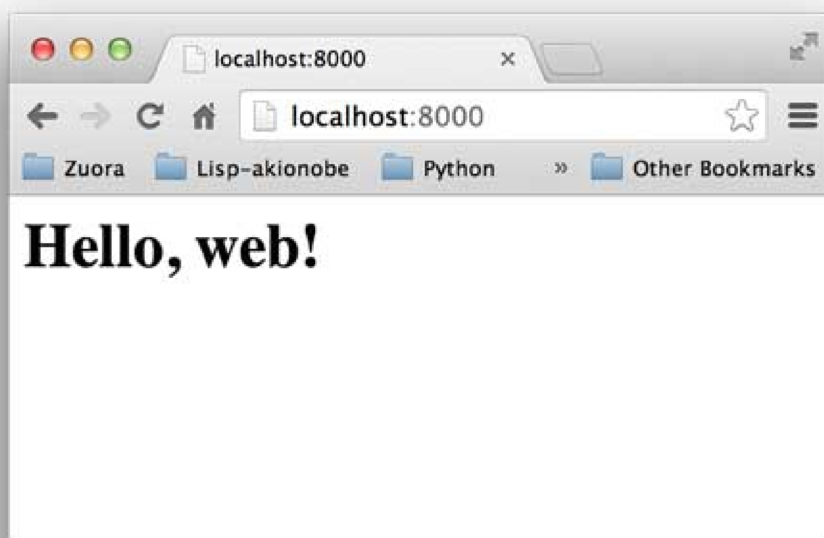
# 创建一个服务器，IP地址为空，端口是8000，处理函数是application:
httpd = make_server('', 8000, application)
print "Serving HTTP on port 8000..."
# 开始监听HTTP请求:
httpd.serve_forever()
```

确保以上两个文件在同一个目录下，然后在命令行输入 `python server.py` 来启动WSGI服务器：

```
michael — Python — 80×24
Michaels-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
```

注意：如果 `8000` 端口已被其他程序占用，启动将失败，请修改成其他端口。

启动成功后，打开浏览器，输入 `http://localhost:8000/` ，就可以看到结果了：



在命令行可以看到wsgiref打印的log信息：

```
Michaels-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
127.0.0.1 - - [14/May/2014 11:01:11] "GET / HTTP/1.1" 200 20
127.0.0.1 - - [14/May/2014 11:01:11] "GET /favicon.ico HTTP/1.1" 200 20
```

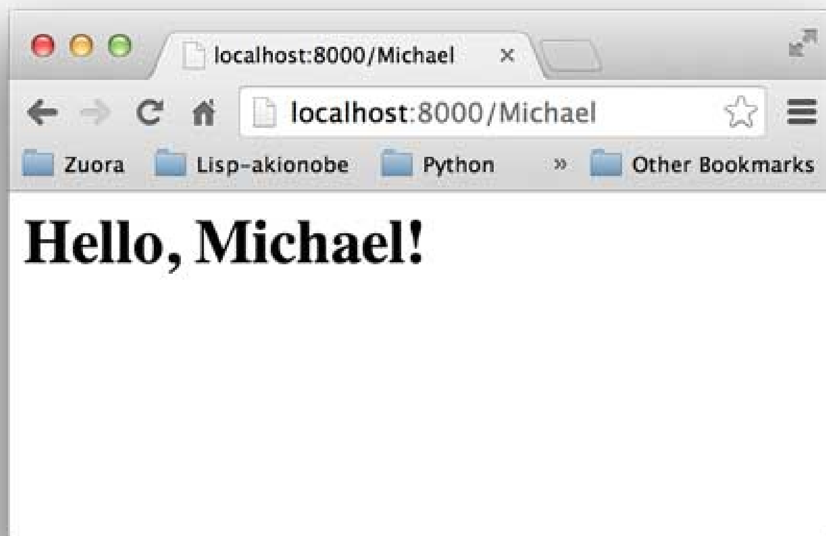
按 `Ctrl+C` 终止服务器。

如果你觉得这个Web应用太简单了，可以稍微改造一下，从 `environ` 里读取 `PATH_INFO` ，这样可以显示更加动态的内容：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return '<h1>Hello, %s!</h1>' % (environ['PATH_INFO'][1:] or 'we
```

你可以在地址栏输入用户名作为URL的一部分，将返回 `Hello, xxx!` ：



是不是有点Web App的感觉了？

小结

无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过 `environ` 获得，HTTP响应的输出都可以通过 `start_response()` 加上函数返回值作为Body。

复杂的Web应用程序，光靠一个WSGI函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

使用Web框架

了解了WSGI框架，我们发现：其实一个Web App，就是写一个WSGI的处理函数，针对每个HTTP请求进行响应。

但是如何处理HTTP请求不是问题，问题是如何处理100个不同的URL。

每一个URL可以对应GET和POST请求，当然还有PUT、DELETE等请求，但是我们通常只考虑最常见的GET和POST请求。

一个最简单的想法是从 `environ` 变量里取出HTTP请求的信息，然后逐个判断：

```
def application(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    if method=='GET' and path=='/':
        return handle_home(environ, start_response)
    if method=='POST' and path='/signin':
        return handle_signin(environ, start_response)
    ...
```

只是这么写下去代码是肯定没法维护了。

代码这么写没法维护的原因是因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级，我们需要在WSGI接口之上能进一步抽象，让我们专注于用一个函数处理一个URL，至于URL到函数的映射，就交给Web框架来做。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。这里我们先不讨论各种Web框架的优缺点，直接选择一个比较流行的Web框架——Flask来使用。

用Flask编写Web App比WSGI接口简单（这不是废话么，要是比WSGI还复杂，用框架干嘛？），我们先用 `easy_install` 或者 `pip` 安装Flask：

```
$ easy_install flask
```

然后写一个 `app.py` ，处理3个URL，分别是：

- `GET /` ：首页，返回 `Home` ；
- `GET /signin` ：登录页，显示登录表单；
- `POST /signin` ：处理登录表单，显示登录结果。

注意噢，同一个URL `/signin` 分别有GET和POST两种请求，映射到两个处理函数中。

Flask通过Python的装饰器在内部自动地把URL和函数给关联起来，所以，我们写出来的代码就像这样：

```
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return '<h1>Home</h1>'

@app.route('/signin', methods=['GET'])
def signin_form():
    return '''<form action="/signin" method="post">
              <p><input name="username"></p>
              <p><input name="password" type="password"></p>
              <p><button type="submit">Sign In</button></p>
              </form>'''

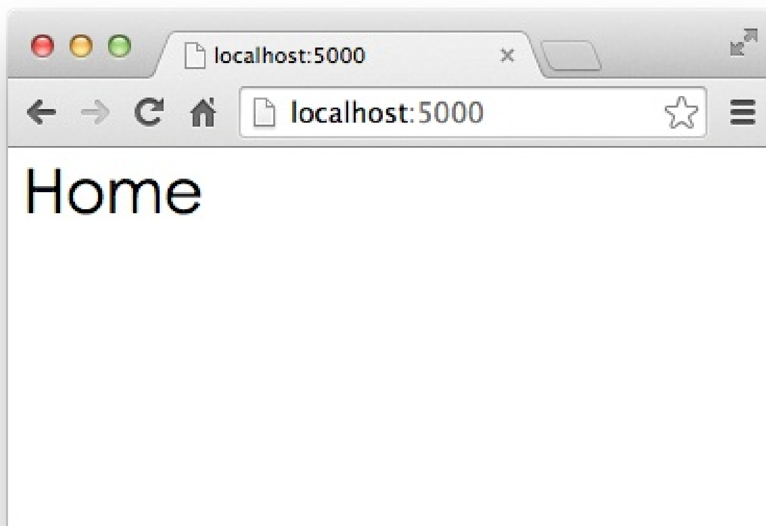
@app.route('/signin', methods=['POST'])
def signin():
    # 需要从request对象读取表单内容：
    if request.form['username']=='admin' and request.form['passwor
        return '<h3>Hello, admin!</h3>'
    return '<h3>Bad username or password.</h3>'

if __name__ == '__main__':
    app.run()
```

运行 `python app.py` ，Flask自带的Server在端口 `5000` 上监听：

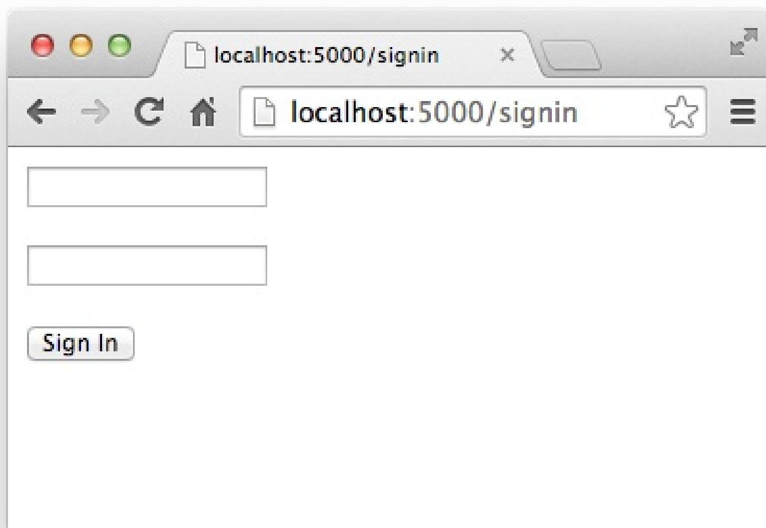
```
$ python app.py
 * Running on http://127.0.0.1:5000/
```

打开浏览器，输入首页地址 `http://localhost:5000/` ：

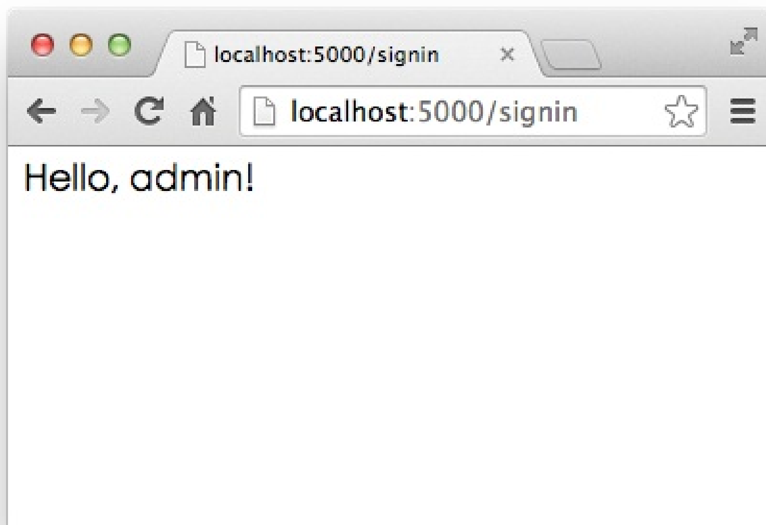


首页显示正确！

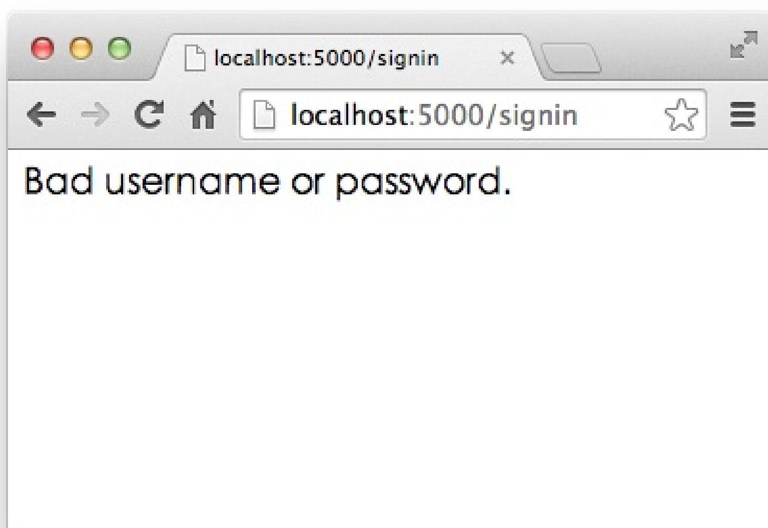
再在浏览器地址栏输入 `http://localhost:5000/signin` ，会显示登录表单：



输入预设的用户名 `admin` 和口令 `password` ，登录成功：



输入其他错误的用户名和口令，登录失败：



实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

除了Flask，常见的Python Web框架还有：

- Django：全能型Web框架；
- web.py：一个小巧的Web框架；
- Bottle：和Flask类似的Web框架；
- Tornado：Facebook的开源异步Web框架。

当然了，因为开发Python的Web框架也不是什么难事，我们后面也会自己开发一个Web框架。

小结

有了Web框架，我们在编写Web应用时，注意力就从WSGI处理函数转移到URL+对应的处理函数，这样，编写Web App就更加简单了。

在编写URL处理函数时，除了配置URL外，从HTTP请求拿到用户数据也是非常重要的。Web框架都提供了自己的API来实现这些功能。Flask通过 `request.form['name']` 来获取表单的内容。

使用模板

Web框架把我们从WSGI中拯救出来了。现在，我们只需要不断地编写函数，带上URL，就可以继续Web App的开发了。

但是，Web App不仅仅是处理逻辑，展示给用户的页面也非常重要。在函数中返回一个包含HTML的字符串，简单的页面还可以，但是，想想新浪首页的6000多行的HTML，你确信能在Python的字符串中正确地写出来么？反正我是做不到。

俗话说得好，不懂前端的Python工程师不是好的产品经理。有Web开发经验的同学都明白，Web App最复杂的部分就在HTML页面。HTML不仅要正确，还要通过CSS美化，再加上复杂的JavaScript脚本来实现各种交互和动画效果。总之，生成HTML页面的难度很大。

由于在Python代码里拼字符串是不现实的，所以，模板技术出现了。

使用模板，我们需要预先准备一个HTML文档，这个HTML文档不是普通的HTML，而是嵌入了一些变量和指令，然后，根据我们传入的数据，替换后，得到最终的HTML，发送给用户：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

Python处理URL的函数就是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量 `{{ name }}` 的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个 `dict` ：

```
{ 'name': 'Michael' }
```

只是因为Python支持关键字参数，很多Web框架允许传入关键字参数，然后，在框架内部组装出一个 `dict` 作为Model。

现在，我们把上次直接输出字符串作为HTML的例子用高端大气上档次的MVC模式改写一下：

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return render_template('home.html')

@app.route('/signin', methods=['GET'])
def signin_form():
    return render_template('form.html')

@app.route('/signin', methods=['POST'])
def signin():
    username = request.form['username']
    password = request.form['password']
    if username=='admin' and password=='password':
        return render_template('signin-ok.html', username=username)
    return render_template('form.html', message='Bad username or pa

if __name__ == '__main__':
    app.run()
```

Flask通过 `render_template()` 函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是jinja2，所以我们先直接安装jinja2：

```
$ easy_install jinja2
```

然后，开始编写jinja2模板：

home.html

用来显示首页的模板：

```
<html>
<head>
  <title>Home</title>
</head>
<body>
  <h1 style="font-style:italic">Home</h1>
</body>
</html>
```

form.html

用来显示登录表单的模板：

```
<html>
<head>
  <title>Please Sign In</title>
</head>
<body>
  {% if message %}
  <p style="color:red">{{ message }}</p>
  {% endif %}
  <form action="/signin" method="post">
    <legend>Please sign in:</legend>
    <p><input name="username" placeholder="Username" value="{{ use
    <p><input name="password" placeholder="Password" type="passwor
    <p><button type="submit">Sign In</button></p>
  </form>
</body>
</html>
```

signin-ok.html

登录成功的模板：

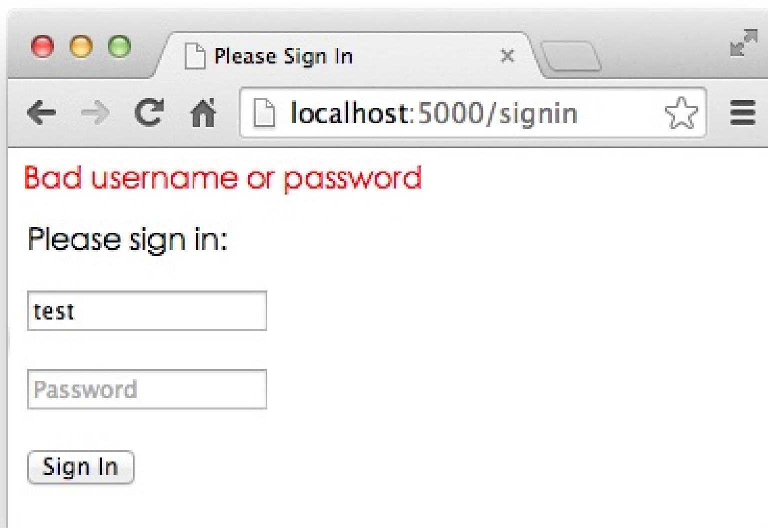
```
<html>
<head>
  <title>Welcome, {{ username }}</title>
</head>
<body>
  <p>Welcome, {{ username }}!</p>
</body>
</html>
```

登录失败的模板呢？我们在 `form.html` 中加了一点条件判断，把 `form.html` 重用为登录失败的模板。

最后，一定要把模板放到正确的 `templates` 目录下， `templates` 和 `app.py` 在同级目录下：



启动 `python app.py`，看看使用模板的页面效果：



通过MVC，我们在Python代码中处理M：Model和C：Controller，而V：View是通过模板处理的，这样，我们就成功地把Python代码和HTML代码最大限度地分离了。

使用模板的另一大好处是，模板改起来很方便，而且，改完保存后，刷新浏览器就能看到最新的效果，这对于调试HTML、CSS和JavaScript的前端工程师来说实在是太重要了。

在Jinja2模板中，我们用 `{{ name }}` 表示一个需要替换的变量。很多时候，还需要循环、条件判断等指令语句，在Jinja2中，用 `{% ... %}` 表示指令。

比如循环输出页码：

```
{% for i in page_list %}
    <a href="/page/{{ i }}">{{ i }}</a>
{% endfor %}
```

如果 `page_list` 是一个list：`[1, 2, 3, 4, 5]`，上面的模板将输出5个超链接。

除了Jinja2，常见的模板还有：

- Mako：用 `<% ... %>` 和 `${xxx}` 的一个模板；
- Cheetah：也是用 `<% ... %>` 和 `${xxx}` 的一个模板；
- Django：Django是一站式框架，内置一个用 `{% ... %}` 和 `{{ xxx }}` 的模板。

小结

有了MVC，我们就分离了Python代码和HTML代码。HTML代码全部放到模板里，写起来更有效率。

协程

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

```
def A():
    print '1'
    print '2'
    print '3'

def B():
    print 'x'
    print 'y'
    print 'z'
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

```
1
2
x
y
3
z
```

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
import time

def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        time.sleep(1)
        r = '200 OK'

def produce(c):
    c.next()
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

if __name__=='__main__':
    c = consumer()
    produce(c)
```

执行结果：

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

注意到consumer函数是一个generator（生成器），把一个consumer传入produce后：

1. 首先调用c.next()启动生成器；
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；
5. produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

gevent

Python通过`yield`提供了对协程的基本支持，但是不完全。而第三方的gevent为Python提供了比较完善的协程支持。

gevent是第三方库，通过greenlet实现协程，其基本思想是：

当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO。

由于切换是在IO操作时自动完成，所以gevent需要修改Python自带的一些标准库，这一过程在启动时通过monkey patch完成：

```
from gevent import monkey; monkey.patch_socket()
import gevent

def f(n):
    for i in range(n):
        print gevent.getcurrent(), i

g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
g1.join()
g2.join()
g3.join()
```

运行结果：

```
<Greenlet at 0x10e49f550: f(5)> 0
<Greenlet at 0x10e49f550: f(5)> 1
<Greenlet at 0x10e49f550: f(5)> 2
<Greenlet at 0x10e49f550: f(5)> 3
<Greenlet at 0x10e49f550: f(5)> 4
<Greenlet at 0x10e49f910: f(5)> 0
<Greenlet at 0x10e49f910: f(5)> 1
<Greenlet at 0x10e49f910: f(5)> 2
<Greenlet at 0x10e49f910: f(5)> 3
<Greenlet at 0x10e49f910: f(5)> 4
<Greenlet at 0x10e49f4b0: f(5)> 0
<Greenlet at 0x10e49f4b0: f(5)> 1
<Greenlet at 0x10e49f4b0: f(5)> 2
<Greenlet at 0x10e49f4b0: f(5)> 3
<Greenlet at 0x10e49f4b0: f(5)> 4
```

可以看到，3个greenlet是依次运行而不是交替运行。

要让greenlet交替运行，可以通过`gevent.sleep()`交出控制权：

```
def f(n):
    for i in range(n):
        print gevent.getcurrent(), i
        gevent.sleep(0)
```

执行结果：

```
<Greenlet at 0x10cd58550: f(5)> 0
<Greenlet at 0x10cd58910: f(5)> 0
<Greenlet at 0x10cd584b0: f(5)> 0
<Greenlet at 0x10cd58550: f(5)> 1
<Greenlet at 0x10cd584b0: f(5)> 1
<Greenlet at 0x10cd58910: f(5)> 1
<Greenlet at 0x10cd58550: f(5)> 2
<Greenlet at 0x10cd58910: f(5)> 2
<Greenlet at 0x10cd584b0: f(5)> 2
<Greenlet at 0x10cd58550: f(5)> 3
<Greenlet at 0x10cd584b0: f(5)> 3
<Greenlet at 0x10cd58910: f(5)> 3
<Greenlet at 0x10cd58550: f(5)> 4
<Greenlet at 0x10cd58910: f(5)> 4
<Greenlet at 0x10cd584b0: f(5)> 4
```

3个greenlet交替运行，

把循环次数改为500000，让它们的运行时间长一点，然后在操作系统的进程管理器中看，线程数只有1个。

当然，实际代码里，我们不会用 `gevent.sleep()` 去切换协程，而是在执行到IO操作时，gevent自动切换，代码如下：

```
from gevent import monkey; monkey.patch_all()
import gevent
import urllib2

def f(url):
    print('GET: %s' % url)
    resp = urllib2.urlopen(url)
    data = resp.read()
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://github.com/'),
])
```

运行结果：

```
GET: https://www.python.org/
GET: https://www.yahoo.com/
GET: https://github.com/
45661 bytes received from https://www.python.org/.
14823 bytes received from https://github.com/.
304034 bytes received from https://www.yahoo.com/.
```

从结果看，3个网络操作是并发执行的，而且结束顺序不同，但只有一个线程。

小结

使用gevent，可以获得极高的并发性能，但gevent只能在Unix/Linux下运行，在Windows下不保证正常安装和运行。

由于gevent是基于IO切换的协程，所以最神奇的是，我们编写的Web App代码，不需要引入gevent的包，也不需要改任何代码，仅仅在部署的时候，用一个支持gevent的WSGI服务器，立刻就获得了数倍的性能提升。具体部署方式可以参考后续“实战”-“部署Web App”一节。

实战

看完了教程，是不是有这么一种感觉：看的时候觉得很简单，照着教程敲代码也没啥大问题。

于是准备开始独立写代码，就发现不知道从哪开始下手了。

这种情况是完全正常的。好比学写作文，学的时候觉得简单，写的时候就无从下笔了。

虽然这个教程是面向小白的零基础Python教程，但是我们的目标不是学到60分，而是学到90分。

所以，用Python写一个真正的Web App吧！

目标

我们设定的实战目标是一个Blog网站，包含日志、用户和评论3大部分。

很多童鞋会想，这是不是太简单了？

比如webpy.org上就提供了一个Blog的例子，目测也就100行代码。

但是，这样的页面：

Hello, world

My first web app...

你拿得出手么？

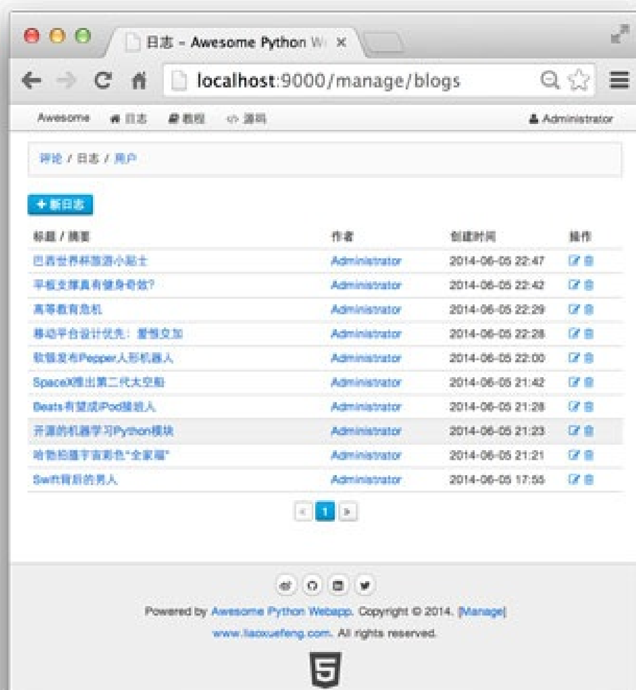
我们要写出用户真正看得上眼的页面，首页长得像这样：


日志 - Awesome Python W
localhost:9000
Awesome
日志
教程
源码
Administrator
巴西世界杯旅游小贴士
发表于10小时前
继续阅读 »
平板支撑真有健身奇效?
发表于10小时前
最近，"平板支撑（Plank）"似乎成了时尚运动的代名词。它被誉为"最实用的减肥动作"，号称每天只需几分钟就能帮你消灭赘肉。
继续阅读 »
高等教育危机
发表于10小时前
一百年前，高等教育的一场技术革命似乎迫在眉睫。作为一种强大的新型通信网络，现代函授系统让大学的校外授课成为可能。
继续阅读 »
移动平台设计优先：爱恨交加
发表于10小时前
如果你仍然认为移动网页设计和应用开发是小众市场，那么你需要改变一下你的思路了。移动并不是趋势，即使将来也不是，它就是当下。
继续阅读 »
软银发布Pepper人形机器人
发表于11小时前
日本软银公司CEO孙正义周四在东京举行的新闻发布会上表示，该公司将正式涉足机器人业务。
继续阅读 »
友情链接
编程
读书
Python教程
Git教程

评论区：



还有极其强大的后台管理页面：



是不是一下子变得高端大气上档次了？

项目名称

必须是高端大气上档次的名称，命名为 `awesome-python-webapp` 。

项目计划

项目计划开发周期为16天。每天，你需要完成教程中的内容。如果你觉得编写代码难度实在太大，可以参考一下当天在GitHub上的代码。

第N天的代码在 `https://github.com/michaelliao/awesome-python-webapp/tree/day-N` 上。比如第1天就是：

https://github.com/michaelliao/awesome-python-webapp/tree/day-01

以此类推。

要预览 `awesome-python-webapp` 的最终页面效果，请猛击：

awesome.liaoxuefeng.com

Day 1 - 搭建开发环境

搭建开发环境

首先，确认系统安装的Python版本是2.7.x：

```
$ python --version
Python 2.7.5
```

然后，安装开发Web App需要的第三方库：

前端模板引擎jinja2：

```
$ easy_install jinja2
```

MySQL 5.x数据库，从官方网站下载并安装，安装完毕后，请务必牢记root口令。为避免遗忘口令，建议直接把root口令设置为 `password` ；

MySQL的Python驱动程序mysql-connector-python：

```
$ easy_install mysql-connector-python
```

项目结构

选择一个工作目录，然后，我们建立如下的目录结构：

```
awesome-python-webapp/   <-- 根目录
|
+- backup/               <-- 备份目录
|
+- conf/                 <-- 配置文件
|
+- dist/                 <-- 打包目录
|
+- www/                  <-- Web目录，存放.py文件
|  |
|  +- static/            <-- 存放静态文件
|  |
|  +- templates/         <-- 存放模板文件
|
+- LICENSE               <-- 代码LICENSE
```

创建好项目的目录结构后，建议同时建立Git仓库并同步至GitHub，保证代码修改的安全。

要了解Git和GitHub的用法，请移步Git教程。

开发工具

自备，推荐用Sublime Text。

Day 2 - 编写数据库模块

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python-app中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

此外，在一个Web App中，有多个用户会同时访问，系统以多进程或多线程模式来处理每个用户的请求。假设以多线程为例，每个线程在访问数据库时，都必须创建仅属于自身的连接，对别的线程不可见，否则，就会造成数据库操作混乱。

所以，我们还要创建一个简单可靠的数据库访问模型，在一个线程中，能既安全又简单地操作数据库。

为什么不选择SQLAlchemy？SQLAlchemy太庞大，过度地面向对象设计导致API太复杂。

所以我们决定自己设计一个封装基本的SELECT、INSERT、UPDATE和DELETE操作的db模块： `transwarp.db` 。

设计db接口

设计底层模块的原则是，根据上层调用者设计简单易用的API接口，然后，实现模块内部代码。

假设 `transwarp.db` 模块已经编写完毕，我们希望以这样的方式来调用它：

首先，初始化数据库连接信息，通过 `create_engine()` 函数：

```
from transwarp import db
db.create_engine(user='root', password='password', database='test',
```

然后，就可以直接操作SQL了。

如果需要做一个查询，可以直接调用 `select()` 方法，返回的是list，每一个元素是用dict表示的对应的行：

```
users = db.select('select * from user')
# users =>
# [
#     { "id": 1, "name": "Michael"},
#     { "id": 2, "name": "Bob"},
#     { "id": 3, "name": "Adam"}
# ]
```

如果要执行INSERT、UPDATE或DELETE操作，执行 `update()` 方法，返回受影响的行数：

```
n = db.update('insert into user(id, name) values(?, ?)', 4, 'Jack')
```

`update()` 函数签名为：

```
update(sql, *args)
```

统一用 `?` 作为占位符，并传入可变参数来绑定，从根本上避免SQL注入攻击。

每个 `select()` 或 `update()` 调用，都隐含地自动打开并关闭了数据库连接，这样，上层调用者就完全不必关心数据库底层连接。

但是，如果要在一个数据库连接里执行多个SQL语句怎么办？我们用一个with语句实现：

```
with db.connection():
    db.select('...')
    db.update('...')
    db.update('...')
```

如果要在一个数据库事务中执行多个SQL语句怎么办？我们还是用一个with语句实现：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

实现db模块

由于模块是全局对象，模块变量是全局唯一变量，所以，有两个重要的模块变量：

```
# db.py

# 数据库引擎对象:
class _Engine(object):
    def __init__(self, connect):
        self._connect = connect
    def connect(self):
        return self._connect()

engine = None

# 持有数据库连接的上下文对象:
class _DbCtx(threading.local):
    def __init__(self):
        self.connection = None
        self.transactions = 0

    def is_init(self):
        return not self.connection is None

    def init(self):
        self.connection = _LasyConnection()
        self.transactions = 0

    def cleanup(self):
        self.connection.cleanup()
        self.connection = None

    def cursor(self):
        return self.connection.cursor()

_db_ctx = _DbCtx()
```

由于 `_db_ctx` 是 `threadlocal` 对象，所以，它持有的数据库连接对于每个线程看到的都是不一样的。任何一个线程都无法访问到其他线程持有的数据库连接。

有了这两个全局变量，我们继续实现数据库连接的上下文，目的是自动获取和释放连接：

```
class _ConnectionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_cleanup = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_cleanup = True
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        if self.should_cleanup:
            _db_ctx.cleanup()

def connection():
    return _ConnectionCtx()
```

定义了 `__enter__()` 和 `__exit__()` 的对象可以用于with语句，确保任何情况下 `__exit__()` 方法可以被调用。

把 `_ConnectionCtx` 的作用域作用到一个函数调用上，可以这么写：

```
with connection():
    do_some_db_operation()
```

但是更简单的写法是写个@decorator：

```
@with_connection
def do_some_db_operation():
    pass
```

这样，我们实现 `select()` 、 `update()` 方法就更简单了：

```
@with_connection
def select(sql, *args):
    pass

@with_connection
def update(sql, *args):
    pass
```

注意到Connection对象是存储在 `_DbCtx` 这个 `threadlocal` 对象里的，因此，嵌套使用 `with connection()` 也没有问题。 `_DbCtx` 永远检测当前是否已存在Connection，如果存在，直接使用，如果不存在，则打开一个新的Connection。

对于transaction也是类似的， `with transaction()` 定义了一个数据库事务：

```
with db.transaction():
    db.select('...')
    db.update('...')
    db.update('...')
```

函数作用域的事务也有一个简化的@decorator：

```
@with_transaction
def do_in_transaction():
    pass
```

事务也可以嵌套，内层事务会自动合并到外层事务中，这种事务模型足够满足99%的需求。

事务嵌套比Connection嵌套复杂一点，因为事务嵌套需要计数，每遇到一层嵌套就+1，离开一层嵌套就-1，最后到0时提交事务：

```
class _TransactionCtx(object):
    def __enter__(self):
        global _db_ctx
        self.should_close_conn = False
        if not _db_ctx.is_init():
            _db_ctx.init()
            self.should_close_conn = True
        _db_ctx.transactions = _db_ctx.transactions + 1
        return self

    def __exit__(self, exctype, excvalue, traceback):
        global _db_ctx
        _db_ctx.transactions = _db_ctx.transactions - 1
        try:
            if _db_ctx.transactions==0:
                if exctype is None:
                    self.commit()
                else:
                    self.rollback()
        finally:
            if self.should_close_conn:
                _db_ctx.cleanup()

    def commit(self):
        global _db_ctx
        try:
            _db_ctx.connection.commit()
        except:
            _db_ctx.connection.rollback()
            raise

    def rollback(self):
        global _db_ctx
        _db_ctx.connection.rollback()
```

最后，把 `select()` 和 `update()` 方法实现了，db模块就完成了。

Day 3 - 编写ORM

有了db模块，操作数据库直接写SQL就很方便。但是，我们还缺少ORM。如果有了ORM，就可以用类似这样的语句获取User对象：

```
user = User.get('123')
```

而不是写SQL然后再转换成User对象：

```
u = db.select_one('select * from users where id=?', '123')
user = User(**u)
```

所以我们开始编写ORM模块：`transwarp.orm`。

设计ORM接口

和设计db模块类似，设计ORM也是从上层调用者角度来设计。

我们先考虑如何定义一个User对象，然后把数据库表`users`和它关联起来。

```
from transwarp.orm import Model, StringField, IntegerField

class User(Model):
    __table__ = 'users'
    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在`User`类中的`__table__`、`id`和`name`是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述`User`对象和表的映射关系，而实例属性必须通过`__init__()`方法去初始化，所以两者互不干扰：

```
# 创建实例:
user = User(id=123, name='Michael')
# 存入数据库:
user.insert()
```

实现ORM模块

有了定义，我们就可以开始实现ORM模块。

首先要定义的是所有ORM映射的基类 `Model` ：

```
class Model(dict):
    __metaclass__ = ModelMetaclass

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute 

    def __setattr__(self, key, value):
        self[key] = value
```

`Model` 从 `dict` 继承，所以具备所有 `dict` 的功能，同时又实现了特殊方法 `__getattr__()` 和 `__setattr__()` ，所以又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

`Model` 只是一个基类，如何将具体的子类如 `User` 的映射信息读取出来呢？答案就是通过metaclass： `ModelMetaclass` ：

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        mapping = ... # 读取cls的Field字段
        primary_key = ... # 查找primary_key字段
        __table__ = cls.__talbe__ # 读取cls的__table__字段
        # 给cls增加一些字段：
        attrs['__mapping__'] = mapping
        attrs['__primary_key__'] = __primary_key__
        attrs['__table__'] = __table__
        return type.__new__(cls, name, bases, attrs)
```

这样，任何继承自 `Model` 的类（比如 `User` ），会自动通过 `ModelMetaclass` 扫描映射关系，并存储到自身的class中。

然后，我们往 `Model` 类添加class方法，就可以让所有子类调用class方法：

```
class Model(dict):

    ...

    @classmethod
    def get(cls, pk):
        d = db.select_one('select * from %s where %s=?' % (cls.__ta
        return cls(**d) if d else None
```

`User` 类就可以通过类方法实现主键查找：

```
user = User.get('123')
```

往 `Model` 类添加实例方法，就可以让所有子类调用实例方法：

```
class Model(dict):

    ...

    def insert(self):
        params = {}
        for k, v in self.__mappings__.iteritems():
            params[v.name] = getattr(self, k)
        db.insert(self.__table__, **params)
        return self
```

这样，就可以把一个 `User` 实例存入数据库：

```
user = User(id=123, name='Michael')
user.insert()
```

最后一步是完善ORM，对于查找，我们可以实现以下方法：

- find_first()
- find_all()
- find_by()

对于count，可以实现：

- count_all()
- count_by()

以及 `update()` 和 `delete()` 方法。

最后看看我们实现的ORM模块一共多少行代码？加上注释和doctest才仅仅300多行。用Python写一个ORM是不是很容易呢？

Day 4 - 编写Model

有了ORM，我们就可以把Web App需要的3个表用`Model`表示出来：

```
import time, uuid

from transwarp.db import next_id
from transwarp.orm import Model, StringField, BooleanField, FloatFi

class User(Model):
    __table__ = 'users'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
    email = StringField(updatable=False, ddl='varchar(50)')
    password = StringField(ddl='varchar(50)')
    admin = BooleanField()
    name = StringField(ddl='varchar(50)')
    image = StringField(ddl='varchar(500)')
    created_at = FloatField(updatable=False, default=time.time)

class Blog(Model):
    __table__ = 'blogs'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
    user_id = StringField(updatable=False, ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(updatable=False, default=time.time)

class Comment(Model):
    __table__ = 'comments'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
    blog_id = StringField(updatable=False, ddl='varchar(50)')
    user_id = StringField(updatable=False, ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(updatable=False, default=time.time)
```

在编写ORM时，给一个Field增加一个`default`参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用`insert()`时自动计算。

例如，主键`id`的缺省值是函数`next_id`，创建时间`created_at`的缺省值是函数`time.time`，可以自动设置当前日期和时间。

日期和时间用`float`类型存储在数据库中，而不是`datetime`类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个`float`到`str`的转换，也非常容易。

初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

```
-- schema.sql

drop database if exists awesome;

create database awesome;

use awesome;

grant select, insert, update, delete on awesome.* to 'www-data'@'lo

create table users (
    `id` varchar(50) not null,
    `email` varchar(50) not null,
    `password` varchar(50) not null,
    `admin` bool not null,
    `name` varchar(50) not null,
    `image` varchar(500) not null,
    `created_at` real not null,
    unique key `idx_email` (`email`),
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table blogs (
```

```
    `id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `name` varchar(50) not null,
    `summary` varchar(200) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table comments (
    `id` varchar(50) not null,
    `blog_id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;
```

如果表的数量很多，可以从 `Model` 对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

```
$ mysql -u root -p < schema.sql
```

我们就完成了数据库表的初始化。

编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于 `User` 对象，我们就可以做如下操作：

```
# test_db.py

from models import User, Blog, Comment

from transwarp import db

db.create_engine(user='www-data', password='www-data', database='aw

u = User(name='Test', email='test@example.com', password='123456789

u.insert()

print 'new user id:', u.id

u1 = User.find_first('where email=?', 'test@example.com')
print 'find user\'s name:', u1.name

u1.delete()

u2 = User.find_first('where email=?', 'test@example.com')
print 'find user:', u2
```

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

Day 5 - 编写Web框架

在正式开始Web开发前，我们需要编写一个Web框架。

为什么不选择一个现成的Web框架而是自己从头开发呢？我们来考察一下现有的流行的Web框架：

Django：一站式开发框架，但不利于定制化；

web.py：使用类而不是更简单的函数来处理URL，并且URL映射是单独配置的；

Flask：使用@decorator的URL路由不错，但框架对应用程序的代码入侵太强；

bottle：缺少根据URL模式进行拦截的功能，不利于做权限检查。

所以，我们综合几种框架的优点，设计一个简单、灵活、入侵性极小的Web框架。

设计Web框架

一个简单的URL框架应该允许以@decorator方式直接把URL映射到函数上：

```
# 首页:
@get('/')
def index():
    return '<h1>Index page</h1>'

# 带参数的URL:
@get('/user/:id')
def show_user(id):
    user = User.get(id)
    return 'hello, %s' % user.name
```

有没有@decorator不改变函数行为，也就是说，Web框架的API入侵性很小，你可以直接测试函数 `show_user(id)` 而不需要启动Web服务器。

函数可以返回 `str` 、 `unicode` 以及 `iterator` ，这些数据可以直接作为字符串返回给浏览器。

其次，Web框架要支持URL拦截器，这样，我们就可以根据URL做权限检查：

```
@interceptor('/manage/')
def check_manage_url(next):
    if current_user.isAdmin():
        return next()
    else:
        raise seeother('/signin')
```

拦截器接受一个 `next` 函数，这样，一个拦截器可以决定调用 `next()` 继续处理请求还是直接返回。

为了支持MVC，Web框架需要支持模板，但是我们不限定使用哪一种模板，可以选择jinja2，也可以选择mako、Cheetah等等。

要统一模板的接口，函数可以返回 `dict` 并配合@view来渲染模板：

```
@view('index.html')
@get('/')
def index():
    return dict(blogs=get_recent_blogs(), user=get_current_user())
```

如果需要从form表单或者URL的querystring获取用户输入的数据，就需要访问 `request` 对象，如果要设置特定的Content-Type、设置Cookie等，就需要访问 `response` 对象。 `request` 和 `response` 对象应该从一个唯一的ThreadLocal中获取：

```
@get('/test')
def test():
    input_data = ctx.request.input()
    ctx.response.content_type = 'text/plain'
    ctx.response.set_cookie('name', 'value', expires=3600)
    return 'result'
```

最后，如果需要重定向、或者返回一个HTTP错误码，最好的方法是直接抛出异常，例如，重定向到登陆页：

```
raise seeother('/signin')
```

返回404错误：

```
raise notfound()
```

基于以上接口，我们就可以实现Web框架了。

实现Web框架

最基本的几个对象如下：

```
# transwarp/web.py

# 全局ThreadLocal对象：
ctx = threading.local()

# HTTP错误类:
class HttpError(Exception):
    pass

# request对象:
class Request(object):
    # 根据key返回value:
    def get(self, key, default=None):
        pass

    # 返回key-value的dict:
    def input(self):
        pass

    # 返回URL的path:
    @property
    def path_info(self):
        pass

    # 返回HTTP Headers:
    @property
    def headers(self):
        pass
```

```
    # 根据key返回Cookie value:
    def cookie(self, name, default=None):
        pass

# response对象:
class Response(object):
    # 设置header:
    def set_header(self, key, value):
        pass

    # 设置Cookie:
    def set_cookie(self, name, value, max_age=None, expires=None, p
        pass

    # 设置status:
    @property
    def status(self):
        pass
    @status.setter
    def status(self, value):
        pass

# 定义GET:
def get(path):
    pass

# 定义POST:
def post(path):
    pass

# 定义模板:
def view(path):
    pass

# 定义拦截器:
def interceptor(pattern):
    pass

# 定义模板引擎:
```

```
class TemplateEngine(object):
    def __call__(self, path, model):
        pass

# 缺省使用jinja2:
class Jinja2TemplateEngine(TemplateEngine):
    def __init__(self, templ_dir, **kw):
        from jinja2 import Environment, FileSystemLoader
        self._env = Environment(loader=FileSystemLoader(templ_dir),

    def __call__(self, path, model):
        return self._env.get_template(path).render(**model).encode(
```

把上面的定义填充完毕，我们就只剩下一件事情：定义全局 `WSGIApplication` 的类，实现WSGI接口，然后，通过配置启动，就完成了整个Web框架的工作。

设计 `WSGIApplication` 要充分考虑开发模式（Development Mode）和产品模式（Production Mode）的区分。在产品模式下， `WSGIApplication` 需要直接提供WSGI接口给服务器，让服务器调用该接口，而在开发模式下，我们更希望能通过 `app.run()` 直接启动服务器进行开发调试：

```
wsgi = WSGIApplication()
if __name__ == '__main__':
    wsgi.run()
else:
    application = wsgi.get_wsgi_application()
```

因此， `WSGIApplication` 定义如下：

```
class WSGIApplication(object):
    def __init__(self, document_root=None, **kw):
        pass

    # 添加一个URL定义:
    def add_url(self, func):
        pass

    # 添加一个Interceptor定义:
    def add_interceptor(self, func):
        pass

    # 设置TemplateEngine:
    @property
    def template_engine(self):
        pass

    @template_engine.setter
    def template_engine(self, engine):
        pass

    # 返回WSGI处理函数:
    def get_wsgi_application(self):
        def wsgi(env, start_response):
            pass
        return wsgi

    # 开发模式下直接启动服务器:
    def run(self, port=9000, host='127.0.0.1'):
        from wsgiref.simple_server import make_server
        server = make_server(host, port, self.get_wsgi_application(
        server.serve_forever()
```

把 `WSGIApplication` 类填充完毕，我们就得到了一个完整的Web框架。

Day 6 - 添加配置文件

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的 `.properties` 或者 `.yaml` 等配置文件。

默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置文件命名为 `config_default.py` ：

```
# config_default.py

configs = {
    'db': {
        'host': '127.0.0.1',
        'port': 3306,
        'user': 'www-data',
        'password': 'www-data',
        'database': 'awesome'
    },
    'session': {
        'secret': 'AwEsOmE'
    }
}
```

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的host等信息，直接修改 `config_default.py` 不是一个好办法，更好的方法是编写一个 `config_override.py` ，用来覆盖某些默认设置：

```
# config_override.py

configs = {
    'db': {
        'host': '192.168.0.100'
    }
}
```

把 `config_default.py` 作为开发环境的标准配置，把 `config_override.py` 作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从 `config_override.py` 读取。为了简化读取配置文件，可以把所有配置读取到统一的 `config.py` 中：

```
# config.py
configs = config_default.configs

try:
    import config_override
    configs = merge(configs, config_override.configs)
except ImportError:
    pass
```

这样，我们就完成了App的配置。

Day 7 - 编写MVC

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的@decorator和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

```
# urls.py
from transwarp.web import get, view
from models import User, Blog, Comment

@view('test_users.html')
@get('/')
def test_users():
    users = User.find_all()
    return dict(users=users)
```

`@view` 指定的模板文件是 `test_users.html` ，所以我们在模板的根目录 `templates` 下创建 `test_users.html` ：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test users - Awesome Python Webapp</title>
</head>
<body>
    <h1>All users</h1>
    {% for u in users %}
    <p>{{ u.name }} / {{ u.email }}</p>
    {% endfor %}
</body>
</html>
```

接下来，我们创建一个Web App的启动文件 `wsgiapp.py` ，负责初始化数据库、初始化Web框架，然后加载 `urls.py` ，最后启动Web服务：

```
# wsgiapp.py
import logging; logging.basicConfig(level=logging.INFO)
import os

from transwarp import db
from transwarp.web import WSGIApplication, Jinja2TemplateEngine

from config import configs

# 初始化数据库:
db.create_engine(**configs.db)

# 创建一个WSGIApplication:
wsgi = WSGIApplication(os.path.dirname(os.path.abspath(__file__)))
# 初始化jinja2模板引擎:
template_engine = Jinja2TemplateEngine(os.path.join(os.path.dirname
wsgi.template_engine = template_engine

# 加载带有@get/@post的URL处理函数:
import urls
wsgi.add_module(urls)

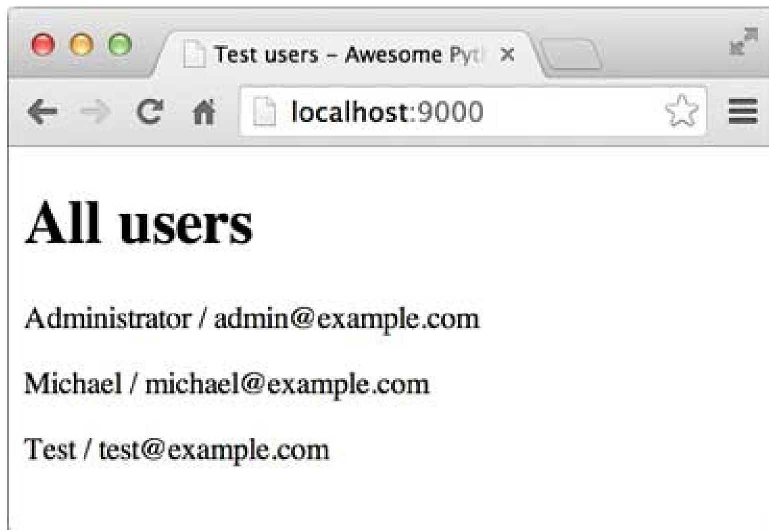
# 在9000端口上启动本地测试服务器:
if __name__ == '__main__':
    wsgi.run(9000)
```

如果一切顺利，可以用命令行启动Web服务器：

```
$ python wsgiapp.py
```

然后，在浏览器中访问 `http://localhost:9000/` 。

如果数据库的 `users` 表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给 `users` 表添加几条记录，然后再访问：


Test users - Awesome Pyt
localhost:9000
All users
Administrator / admin@example.com
Michael / michael@example.com
Test / test@example.com

Day 8 - 构建前端

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择uikit这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从uikit首页下载打包的资源文件。

所有的静态资源文件我们统一放到 `www/static` 目录下，并按照类别归类：

```
static/
+- css/
|  +- addons/
|  |  +- uikit.addons.min.css
|  |  +- uikit.almost-flat.addons.min.css
|  |  +- uikit.gradient.addons.min.css
|  +- awesome.css
|  +- uikit.almost-flat.addons.min.css
|  +- uikit.gradient.addons.min.css
|  +- uikit.min.css
+- fonts/
|  +- fontawesome-webfont.eot
|  +- fontawesome-webfont.ttf
|  +- fontawesome-webfont.woff
|  +- FontAwesome.otf
+- js/
   +- awesome.js
   +- html5.js
   +- jquery.min.js
   +- uikit.min.js
```

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

```
<html>
    <% include file="inc_header.html" %>
    <% include file="index_body.html" %>
    <% include file="inc_footer.html" %>
</html>
```

这样，相同的部分 `inc_header.html` 和 `inc_footer.html` 就可以共享。

但是include方法不利于页面整体结构的维护。jinjia2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

```
<!-- base.html -->
<html>
    <head>
        <title>{% block title%} 这里定义了一个名为title的block {% endb
    </head>
    <body>
        {% block content %} 这里定义了一个名为content的block {% endblo
    </body>
</html>
```

对于子模板 `a.html` ，只需要把父模板的 `title` 和 `content` 替换掉：

```
{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}
<h1>Chapter A</h1>
<p>blablabla...</p>
{% endblock %}
```

对于子模板 `b.html`，如法炮制：

```
{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}
<h1>Chapter B</h1>
<ul>
   <li>list 1</li>
   <li>list 2</li>
</ul>
{% endblock %}
```

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板 `__base__.html` 的编写：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    {% block meta %}<!-- block meta  -->{% endblock %}
    <title>{% block title %} ? {% endblock %} - Awesome Python Weba
    <link rel="stylesheet" href="/static/css/uikit.min.css">
    <link rel="stylesheet" href="/static/css/uikit.gradient.min.css
    <link rel="stylesheet" href="/static/css/awesome.css" />
    <script src="/static/js/jquery.min.js"></script>
    <script src="/static/js/md5.js"></script>
```

```
    <script src="/static/js/uikit.min.js"></script>
    <script src="/static/js/awesome.js"></script>
    {% block beforehead %}<!-- before head  -->{% endblock %}
</head>
<body>
    <nav class="uk-navbar uk-navbar-attached uk-margin-bottom">
        <div class="uk-container uk-container-center">
            <a href="/" class="uk-navbar-brand">Awesome</a>
            <ul class="uk-navbar-nav">
                <li data-url="blogs"><a href="/"><i class="uk-icon-
                <li><a target="_blank" href="#"><i class="uk-icon-b
                <li><a target="_blank" href="#"><i class="uk-icon-c
            </ul>
            <div class="uk-navbar-flip">
                <ul class="uk-navbar-nav">
                {% if user %}
                    <li class="uk-parent" data-uk-dropdown>
                        <a href="#0"><i class="uk-icon-user"></i> <
                        <div class="uk-dropdown uk-dropdown-navbar"
                            <ul class="uk-nav uk-nav-navbar">
                                <li><a href="/signout"><i class="uk
                            </ul>
                        </div>
                    </li>
                {% else %}
                    <li><a href="/signin"><i class="uk-icon-sign-in
                    <li><a href="/register"><i class="uk-icon-edit"
                {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <div class="uk-container uk-container-center">
        <div class="uk-grid">
            <!-- content -->
            {% block content %}
            {% endblock %}
            <!-- // content -->
        </div>
```

```
    </div>

    <div class="uk-margin-large-top" style="background-color:#eee;
        <div class="uk-container uk-container-center uk-text-center
            <div class="uk-panel uk-margin-top uk-margin-bottom">
                <p>
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                </p>
                <p>Powered by <a href="#">Awesome Python Webapp</a>
                <p><a href="http://www.liaoxuefeng.com/" target="_b
                <a target="_blank" href="#"><i class="uk-icon-html5
            </div>
        </div>
    </div>
</body>
</html>
```

`__base__.html` 定义的几个block作用如下：

用于子页面定义一些meta，例如rss feed：

```
{% block meta %} ... {% endblock %}
```

覆盖页面的标题：

```
{% block title %} ... {% endblock %}
```

子页面可以在标签关闭前插入JavaScript代码：

```
{% block beforehead %} ... {% endblock %}
```

子页面的content布局和内容：

```
{% block content %}
...
{% endblock %}
```

我们把首页改造一下，从 `__base__.html` 继承一个 `blogs.html` ：

```
{% extends '__base__.html' %}

{% block title %}日志{% endblock %}

{% block content %}

    <div class="uk-width-medium-3-4">
    {% for blog in blogs %}
        <article class="uk-article">
            <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></
            <p class="uk-article-meta">发表于{{ blog.created_at}}</p
            <p>{{ blog.summary }}</p>
            <p><a href="/blog/{{ blog.id }}">继续阅读 <i class="uk-i
        </article>
        <hr class="uk-article-divider">
    {% endfor %}
    </div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
            </ul>
        </div>
    </div>

{% endblock %}
```

相应地，首页URL的处理函数更新如下：

```
@view('blogs.html')
@get('/')
def index():
    blogs = Blog.find_all()
    # 查找登陆用户：
    user = User.find_first('where email=?', 'admin@example.com')
    return dict(blogs=blogs, user=user)
```

往MySQL的 `blogs` 表中手动插入一些数据，我们就可以看到一个真正的首页了。但是Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

```
<p class="uk-article-meta">发表于{{ blog.created_at }}</p>
```

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个 `datetime` 的filter，在模板里用法如下：

```
<p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
```

filter需要在初始化jinja2时设置。修改 `wsgiapp.py` 相关代码如下：

```
# wsgiapp.py:

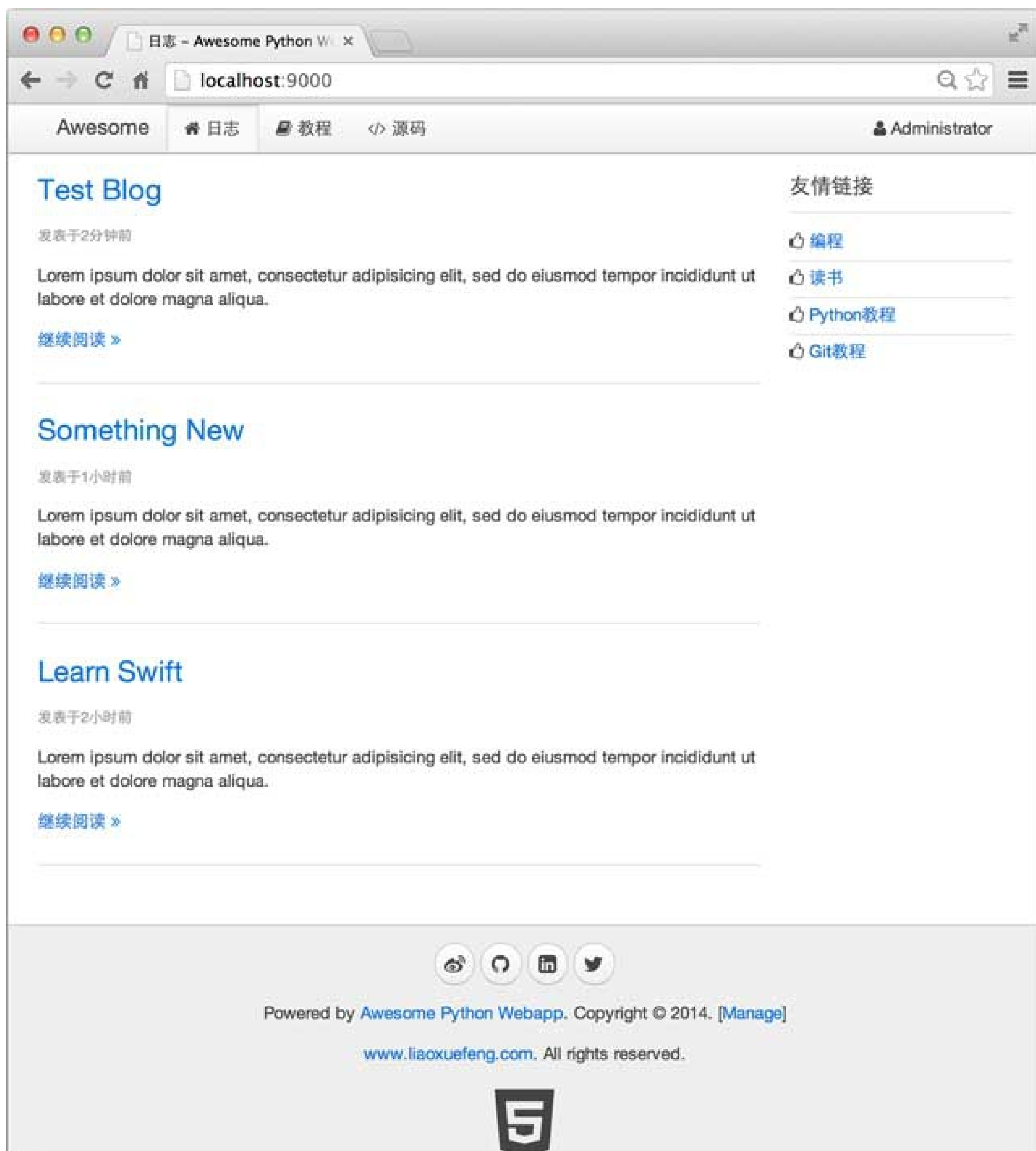
...

# 定义datetime_filter，输入是t，输出是unicode字符串:
def datetime_filter(t):
    delta = int(time.time() - t)
    if delta < 60:
        return u'1分钟前'
    if delta < 3600:
        return u'%s分钟前' % (delta // 60)
    if delta < 86400:
        return u'%s小时前' % (delta // 3600)
    if delta < 604800:
        return u'%s天前' % (delta // 86400)
    dt = datetime.fromtimestamp(t)
    return u'%s年%s月%s日' % (dt.year, dt.month, dt.day)

template_engine = Jinja2TemplateEngine(os.path.join(os.path.dirname
# 把filter添加到jinjia2，filter名称为datetime，filter本身是一个函数对象:
template_engine.add_filter('datetime', datetime_filter)

wsgi.template_engine = template_engine
```

现在，完善的首页显示如下：


日志 - Awesome Python W
localhost:9000
Awesome
日志
教程
源码
Administrator
Test Blog
发表于2分钟前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
Something New
发表于1小时前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
Learn Swift
发表于2小时前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
友情链接
编程
读书
Python教程
Git教程
Powered by Awesome Python Webapp. Copyright © 2014. [Manage]
www.liaoxuefeng.com. All rights reserved.

Day 9 - 编写API

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入`http://localhost:9000/blog/123`，就可以看到id为`123`的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取`http://localhost:9000/api/blogs/123`，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个`@api`来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

```
@api
@get('/api/users')
def api_get_users():
    users = User.find_by('order by created_at desc')
    # 把用户的口令隐藏掉:
    for u in users:
        u.password = '******'
    return dict(users=users)
```

所以，`@api`这个decorator只要编写好了，就可以把任意的URL处理函数变成API调用。

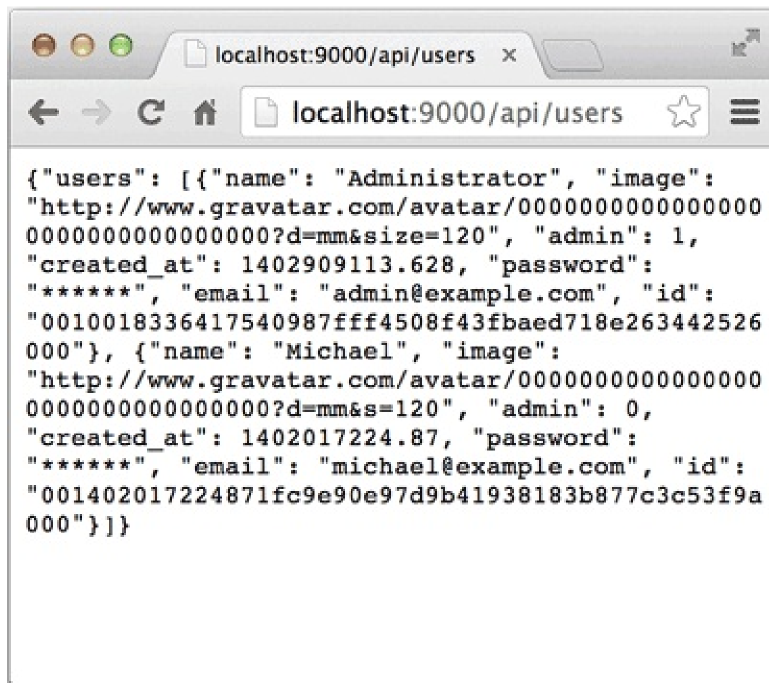
新建一个`apis.py`，编写`@api`负责把函数的返回结果序列化为JSON：

```
def api(func):
    @functools.wraps(func)
    def _wrapper(*args, **kw):
        try:
            r = json.dumps(func(*args, **kw))
        except APIError, e:
            r = json.dumps(dict(error=e.error, data=e.data, message
        except Exception, e:
            r = json.dumps(dict(error='internalerror', data=e.__cla
        ctx.response.content_type = 'application/json'
        return r
    return _wrapper
```

`@api`需要对Error进行处理。我们定义一个`APIError`，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为`internalerror`。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入`http://localhost:9000/api/users`，就可以看到返回的JSON：


localhost:9000/api/users
localhost:9000/api/users
{"users": [{"name": "Administrator", "image":
"http://www.gravatar.com/avatar/0000000000000000
0000000000000000?d=mm&size=120", "admin": 1,
"created_at": 1402909113.628, "password":
"******", "email": "admin@example.com", "id":
"0010018336417540987fff4508f43fbaed718e263442526
000"}, {"name": "Michael", "image":
"http://www.gravatar.com/avatar/0000000000000000
0000000000000000?d=mm&s=120", "admin": 0,
"created_at": 1402017224.87, "password":
"******", "email": "michael@example.com", "id":
"001402017224871fc9e90e97d9b41938183b877c3c53f9a
000"}]}

Day 10 - 用户注册和登录

用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

```
_RE_MD5 = re.compile(r'^[0-9a-f]{32}$')

@api
@post('/api/users')
def register_user():
    i = ctx.request.input(name='', email='', password='')
    name = i.name.strip()
    email = i.email.strip().lower()
    password = i.password
    if not name:
        raise APIValueError('name')
    if not email or not _RE_EMAIL.match(email):
        raise APIValueError('email')
    if not password or not _RE_MD5.match(password):
        raise APIValueError('password')
    user = User.find_first('where email=?', email)
    if user:
        raise APIError('register:failed', 'email', 'Email is alreac
    user = User(name=name, email=email, password=password, image='h
    user.insert()
    return user
```

注意用户口令是客户端传递的经过MD5计算后的32位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

```
{% extends '__base__.html' %}
```

```
{% block title %}注册{% endblock %}

{% block beforehead %}

<script>
function check_form() {
    $('#password').val(CryptoJS.MD5($('#password1').val()).toString
    return true;
}
</script>

{% endblock %}

{% block content %}

<div class="uk-width-2-3">
    <h1>欢迎注册！</h1>
    <form id="form-register" class="uk-form uk-form-stacked" onsubm
        <div class="uk-alert uk-alert-danger uk-hidden"></div>
        <div class="uk-form-row">
            <label class="uk-form-label">名字:</label>
            <div class="uk-form-controls">
                <input name="name" type="text" class="uk-width-1-1"
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">电子邮件:</label>
            <div class="uk-form-controls">
                <input name="email" type="text" class="uk-width-1-1
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">输入口令:</label>
            <div class="uk-form-controls">
                <input id="password1" type="password" class="uk-wid
                <input id="password" name="password" type="hidden">
            </div>
        </div>
        <div class="uk-form-row">
            <label class="uk-form-label">重复口令:</label>
```

```
                <div class="uk-form-controls">
                    <input name="password2" type="password" maxlength='
                </div>
            </div>
            <div class="uk-form-row">
                <button type="submit" class="uk-button uk-button-primar
            </div>
        </form>
    </div>

{% endblock %}
```

这样我们就把用户注册的功能完成了：

用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如MD5），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

```
"用户id" + "过期时间" + MD5("用户id" + "用户口令" + "过期时间" + "Secret
```

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

- 用户id
- 过期时间
- MD5值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

```
MD5("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

并与浏览器cookie中的MD5进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于MD5是一种单向算法，即可以通过原始字符串计算出MD5，但无法通过MD5反推出原始字符串。

所以登录API可以实现如下：

```
@api
@post('/api/authenticate')
def authenticate():
    i = ctx.request.input()
    email = i.email.strip().lower()
    password = i.password
    user = User.find_first('where email=?', email)
    if user is None:
        raise APIError('auth:failed', 'email', 'Invalid email.')
    elif user.password != password:
        raise APIError('auth:failed', 'password', 'Invalid passwor
    max_age = 604800
    cookie = make_signed_cookie(user.id, user.password, max_age)
    ctx.response.set_cookie(_COOKIE_NAME, cookie, max_age=max_age)
    user.password = '******'
    return user

# 计算加密cookie:
def make_signed_cookie(id, password, max_age):
    expires = str(int(time.time() + max_age))
    L = [id, expires, hashlib.md5('%s-%s-%s-%s' % (id, password, ex
    return '-'.join(L)
```

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用拦截器在处理URL之前，把cookie解析出来，并将登录用户绑定到 `ctx.request` 对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

```
@interceptor('/')
def user_interceptor(next):
    user = None
    cookie = ctx.request.cookies.get(_COOKIE_NAME)
    if cookie:
        user = parse_signed_cookie(cookie)
    ctx.request.user = user
    return next()

# 解密cookie:
def parse_signed_cookie(cookie_str):
    try:
        L = cookie_str.split('-')
        if len(L) != 3:
            return None
        id, expires, md5 = L
        if int(expires) < time.time():
            return None
        user = User.get(id)
        if user is None:
            return None
        if md5 != hashlib.md5('%s-%s-%s-%s' % (id, user.password, e
            return None
        return user
    except:
        return None
```

这样，我们就完成了用户注册和登录的功能。

Day 11 - 编写日志创建页

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog：

```
@api
@post('/api/blogs')
def api_create_blog():
    i = ctx.request.input(name='', summary='', content='')
    name = i.name.strip()
    summary = i.summary.strip()
    content = i.content.strip()
    if not name:
        raise APIValueError('name', 'name cannot be empty.')
    if not summary:
        raise APIValueError('summary', 'summary cannot be empty.')
    if not content:
        raise APIValueError('content', 'content cannot be empty.')
    user = ctx.request.user
    blog = Blog(user_id=user.id, user_name=user.name, name=name, su
    blog.insert()
    return blog
```

编写后端Python代码不但很简单，而且非常容易测试，上面的API：`api_create_blog()`本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

```
s = '<html><head><title>'
  + title
  + '</title></head><body>'
  + body
  + '</body></html>'
```

显然这种方式完全不具备可维护性。所以有第二种模板方式：

```
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    {{ body }}
</body>
</html>
```

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的MVVM：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

```
<script>
var blog = {
    name: 'hello',
    summary: 'this is summary',
    content: 'this is content...'
};
</script>
```

View是纯HTML：

```
<form action="/api/blogs" method="post">
    <input name="name">
    <input name="summary">
    <textarea name="content"></textarea>
    <button type="submit">OK</button>
</form>
```

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择Vue这个简单易用的MVVM框架来实现创建Blog的页面`templates/manage_blog_edit.html`：

```
{% extends '__base__.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}

<script>
var
    action = '{{ action }}',
    redirect = '{{ redirect }}';

var vm;

$(function () {
    vm = new Vue({
        el: '#form-blog',
        data: {
            name: '',
            summary: '',
            content: ''
```

```
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                postApi(action, this.$data, function (err, r) {
                    if (err) {
                        alert(err);
                    }
                    else {
                        alert('保存成功！');
                        return location.assign(redirect);
                    }
                });
            }
        }
    });
});
</script>
{% endblock %}

{% block content %}
<div class="uk-width-1-1">
    <form id="form-blog" v-on="submit: submit" class="uk-form uk-fo
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <input v-model="name" class="uk-width-1-1">
            </div>
        </div>
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <textarea v-model="summary" rows="4" class="uk-widt
            </div>
        </div>
        <div class="uk-form-row">
            <div class="uk-form-controls">
                <textarea v-model="content" rows="8" class="uk-widt
            </div>
        </div>
        <div class="uk-form-row">
            <button type="submit" class="uk-button uk-button-primar
```

```
        </div>
    </form>
</div>
{% endblock %}
```

初始化Vue时，我们指定3个参数：

el：根据选择器查找绑定的View，这里是 `#form-blog` ，就是id为 `form-blog` 的DOM，对应的是一个 `<form>` 标签；

data：JavaScript对象表示的Model，我们初始化为 `{ name: '', summary: '', content: ''}` ；

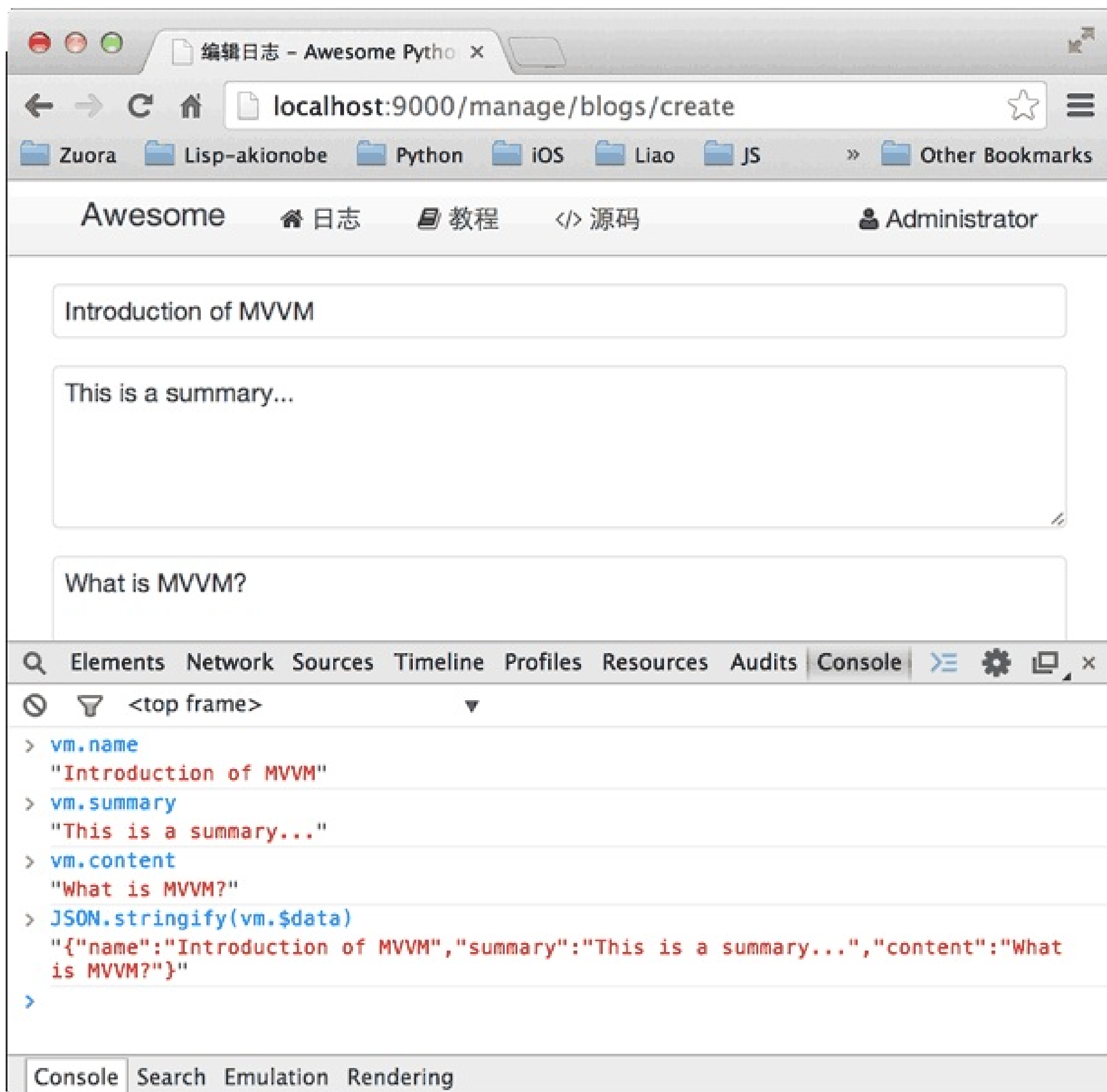
methods：View可以触发的JavaScript函数， `submit` 就是提交表单时触发的函数。

接下来，我们在 `<form>` 标签中，用几个简单的 `v-model` ，就可以让Vue把Model和View关联起来：

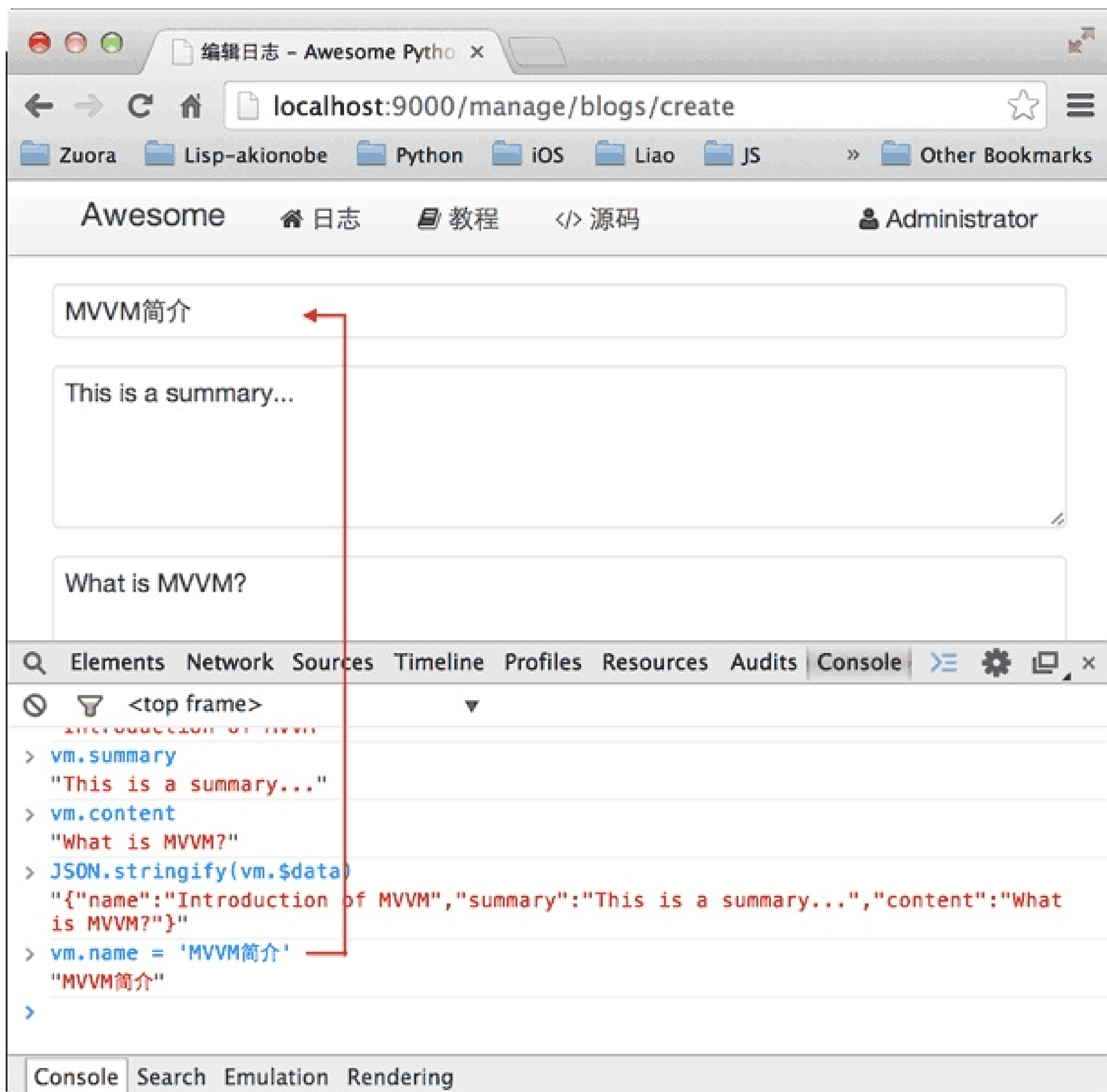
```
<!-- input的value和Model的name关联起来了 -->
<input v-model="name" class="uk-width-1-1">
```

Form表单通过 `<form v-on="submit: submit">` 把提交表单的事件关联到 `submit` 方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过 `vm.name` 访问单个属性，或者通过 `vm.$data` 访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入 `vm.name = 'MVVM简介'` ，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

Day 12 - 编写日志列表页

MVVM模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来：

在 `apis.py` 中定义一个 `Page` 类用于存储分页信息：

```
class Page(object):
    def __init__(self, item_count, page_index=1, page_size=10):
        self.item_count = item_count
        self.page_size = page_size
        self.page_count = item_count // page_size + (1 if item_cour
        if (item_count == 0) or (page_index < 1) or (page_index > s
            self.offset = 0
            self.limit = 0
            self.page_index = 1
        else:
            self.page_index = page_index
            self.offset = self.page_size * (page_index - 1)
            self.limit = self.page_size
        self.has_next = self.page_index < self.page_count
        self.has_previous = self.page_index > 1
```

在 `urls.py` 中实现API：

```
def _get_blogs_by_page():
    total = Blog.count_all()
    page = Page(total, _get_page_index())
    blogs = Blog.find_by('order by created_at desc limit ?,?', page
    return blogs, page

@api
@get('/api/blogs')
def api_get_blogs():
    blogs, page = _get_blogs_by_page()
    return dict(blogs=blogs, page=page)
```

返回模板页面：

```
@view('manage_blog_list.html')
@get('/manage/blogs')
def manage_blogs():
    return dict(page_index=_get_page_index(), user=ctx.request.user
```

模板页面首先通过API： `GET /api/blogs?page=?` 拿到Model：

```
{
    "page": {
        "has_next": true,
        "page_index": 1,
        "page_count": 2,
        "has_previous": false,
        "item_count": 12
    },
    "blogs": [...]
}
```

然后，通过Vue初始化MVVM：

```
<script>
function initVM(data) {
    $('#div-blogs').show();
    var vm = new Vue({
        el: '#div-blogs',
        data: {
            blogs: data.blogs,
            page: data.page
        },
        methods: {
            previous: function () {
                gotoPage(this.page.page_index - 1);
            },
            next: function () {
                gotoPage(this.page.page_index + 1);
            },
            edit_blog: function (blog) {
                location.assign('/manage/blogs/edit/' + blog.id);
            }
        }
    });
}

$(function() {
    getApi('/api/blogs?page={{ page_index }}', function (err, resul
        if (err) {
            return showError(err);
        }
        $('#div-loading').hide();
        initVM(results);
    });
});
</script>
```

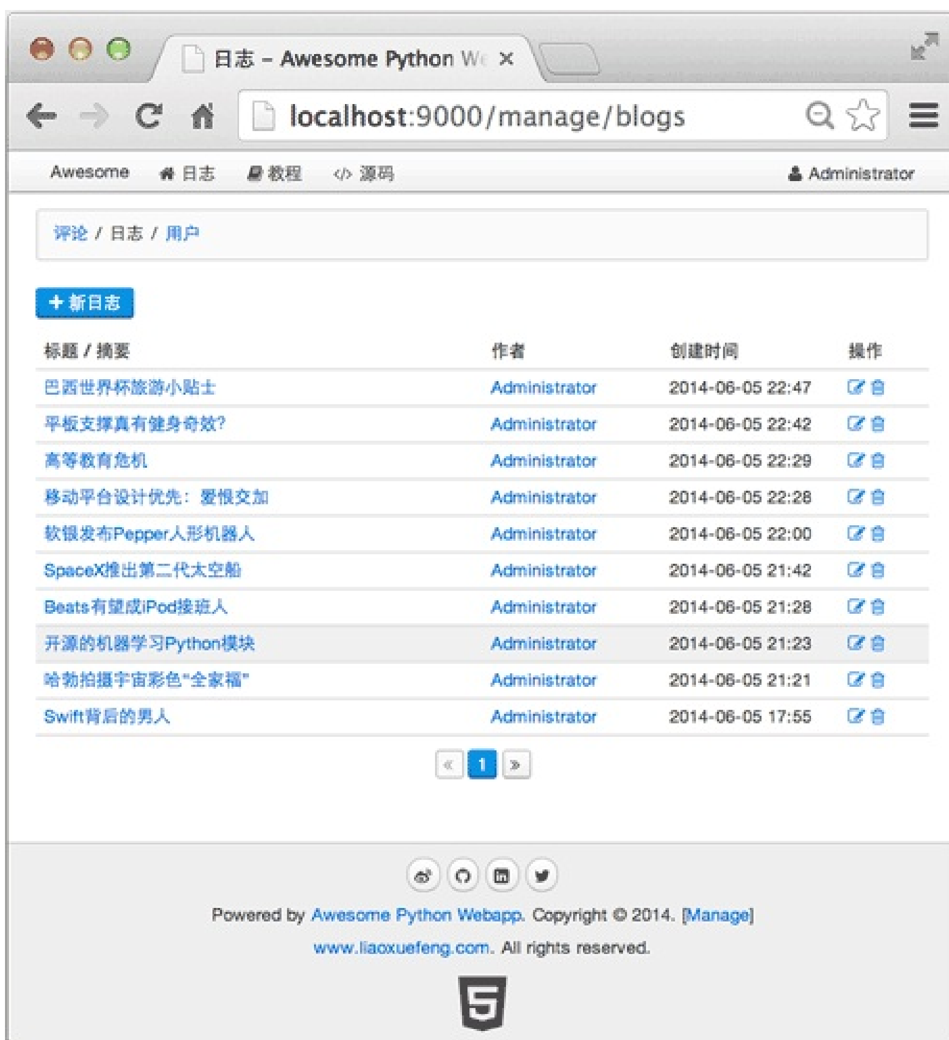
View的容器是 `#div-blogs` ，包含一个table，我们用 `v-repeat` 可以把Model的数组 `blogs` 直接变成多行的 `<tr>` ：

```
<div id="div-blogs" class="uk-width-1-1" style="display:none">
    <table class="uk-table uk-table-hover">
        <thead>
            <tr>
                <th class="uk-width-5-10">标题 / 摘要</th>
                <th class="uk-width-2-10">作者</th>
                <th class="uk-width-2-10">创建时间</th>
                <th class="uk-width-1-10">操作</th>
            </tr>
        </thead>
        <tbody>
            <tr v-repeat="blog: blogs" >
                <td>
                    <a target="_blank" v-attr="href: '/blog/'+blog.
                </td>
                <td>
                    <a target="_blank" v-attr="href: '/user/'+blog.
                </td>
                <td>
                    <span v-text="blog.created_at.toDateTime()"></s
                </td>
                <td>
                    <a href="#0" v-on="click: edit_blog(blog)"><i c
                </td>
            </tr>
        </tbody>
    </table>
    <div class="uk-width-1-1 uk-text-center">
        <ul class="uk-pagination">
            <li v-if="! page.has_previous" class="uk-disabled"><spa
            <li v-if="page.has_previous"><a v-on="click: previous()
            <li class="uk-active"><span v-text="page.page_index"></
            <li v-if="! page.has_next" class="uk-disabled"><span><i
            <li v-if="page.has_next"><a v-on="click: next()" href="
        </ul>
    </div>
</div>
```

往Model的blogs数组中增加一个Blog元素，table就神奇地增加了一行；把blogs数组的某个元素删除，table就神奇地减少了一行。所有复杂的Model-View的映射逻辑全部由MVVM框架完成，我们只需要在HTML中写上 `v-repeat` 指令，就什么都不用管了。

可以把 `v-repeat="blog: blogs"` 看成循环代码，所以，可以在一个 `<tr>` 内部引用循环变量 `blog` 。 `v-text` 和 `v-attr` 指令分别用于生成文本和DOM节点属性。

完整的Blog列表页如下：



Day 13 - 提升开发效率

现在，我们已经把一个Web App的框架完全搭建好了，从后端的API到前端的MVVM，流程已经跑通了。

在继续工作前，注意到每次修改Python代码，都必须在命令行先Ctrl-C停止服务器，再重启，改动才能生效。

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测 `www` 目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序 `pymonitor.py` ，让它启动 `wsgiapp.py` ，并时刻监控 `www` 目录下的代码改动，有改动时，先把当前 `wsgiapp.py` 进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库 `watchdog` 可以利用操作系统的API来监控目录文件的变化，并发送通知。我们先用 `easy_install` 安装：

```
$ easy_install watchdog
```

利用 `watchdog` 接收文件变化的通知，如果是 `.py` 文件，就自动重启 `wsgiapp.py` 进程。

利用Python自带的 `subprocess` 实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

```
#!/usr/bin/env python
import os, sys, time, subprocess
```

```
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

def log(s):
    print '[Monitor] %s' % s

class MyFileSystemEventHander(FileSystemEventHandler):
    def __init__(self, fn):
        super(MyFileSystemEventHander, self).__init__()
        self.restart = fn

    def on_any_event(self, event):
        if event.src_path.endswith('.py'):
            log('Python source file changed: %s' % event.src_path)
            self.restart()

command = ['echo', 'ok']
process = None

def kill_process():
    global process
    if process:
        log('Kill process [%s]...' % process.pid)
        process.kill()
        process.wait()
        log('Process ended with code %s.' % process.returncode)
        process = None

def start_process():
    global process, command
    log('Start process %s...' % ' '.join(command))
    process = subprocess.Popen(command, stdin=sys.stdin, stdout=sys

def restart_process():
    kill_process()
    start_process()

def start_watch(path, callback):
    observer = Observer()
```

```
    observer.schedule(MyFileSystemEventHander(restart_process), pat
    observer.start()
    log('Watching directory %s...' % path)
    start_process()
    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()

if __name__ == '__main__':
    argv = sys.argv[1:]
    if not argv:
        print('Usage: ./pymonitor your-script.py')
        exit(0)
    if argv[0]!='python':
        argv.insert(0, 'python')
    command = argv
    path = os.path.abspath('.')
    start_watch(path, None)
```

一共50行左右的代码，就实现了Debug模式的自动重新加载。用下面的命令启动服务器：

```
$ python pymonitor.py wsgiapp.py
```

或者给 `pymonitor.py` 加上可执行权限，启动服务器：

```
$ ./pymonitor.py wsgiapp.py
```

在编辑器中打开一个py文件，修改后保存，看看命令行输出，是不是自动重启了服务器：

```
$ ./pymonitor.py wsgiapp.py
[Monitor] Watching directory /Users/michael/Github/awesome-python-w
[Monitor] Start process python wsgiapp.py...
...
INFO:root:application (/Users/michael/Github/awesome-python-webapp/
[Monitor] Python source file changed: /Users/michael/Github/awesome
[Monitor] Kill process [2747]...
[Monitor] Process ended with code -9.
[Monitor] Start process python wsgiapp.py...
...
INFO:root:application (/Users/michael/Github/awesome-python-webapp/
```

现在，只要一保存代码，就可以刷新浏览器看到效果，大大提升了开发效率。

Day 14 - 完成Web App

在Web App框架和基本流程跑通后，剩下的工作全部是体力活了：在Debug开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：

对URL `/manage/` 进行拦截，检查当前用户是否是管理员身份：

```
@interceptor('/manage/')
def manage_interceptor(next):
    user = ctx.request.user
    if user and user.admin:
        return next()
    raise seeother('/signin')
```

后端API包括：

- 获取日志：GET /api/blogs
- 创建日志：POST /api/blogs
- 修改日志：POST /api/blogs/:blog_id
- 删除日志：POST /api/blogs/:blog_id/delete
- 获取评论：GET /api/comments
- 创建评论：POST /api/blogs/:blog_id/comments
- 删除评论：POST /api/comments/:comment_id/delete
- 创建新用户：POST /api/users
- 获取用户：GET /api/users

管理页面包括：

- 评论列表页：GET /manage/comments
- 日志列表页：GET /manage/blogs
- 创建日志页：GET /manage/blogs/create

- 修改日志页：GET /manage/blogs/
- 用户列表页：GET /manage/users

用户浏览页面包括：

- 注册页：GET /register
- 登录页：GET /signin
- 注销页：GET /signout
- 首页：GET /
- 日志详情页：GET /blog/:blog_id

把所有的功能实现，我们第一个Web App就宣告完成！

Day 15 - 部署Web App

作为一个合格的开发者，在本地环境下完成开发还远远不够，我们需要把Web App部署到远程服务器上，这样，广大用户才能访问到网站。

很多做开发的同学把部署这件事情看成是运维同学的工作，这种看法是完全错误的。首先，最近流行DevOps理念，就是说，开发和运维要变成一个整体。其次，运维的难度，其实跟开发质量有很大的关系。代码写得垃圾，运维再好也架不住天天挂掉。最后，DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务？那就得在开发时考虑到这一点。

下面，我们就来把awesome-python-webapp部署到Linux服务器。

搭建Linux服务器

要部署到Linux，首先得有一台Linux服务器。要在公网上体验的同学，可以在Amazon的AWS申请一台EC2虚拟机（免费使用1年），或者使用国内的一些云服务器，一般都提供Ubuntu Server的镜像。想在本地部署的同学，请安装虚拟机，推荐使用VirtualBox。

我们选择的Linux服务器版本是Ubuntu Server 12.04 LTS，原因是apt太简单了。如果你准备使用其他Linux版本，也没有问题。

Linux安装完成后，请确保ssh服务正在运行，否则，需要通过apt安装：

```
$ sudo apt-get install openssh-server
```

有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户的 `.ssh/authorized_keys` 中，这样，就可以通过证书实现无密码连接。

部署方式

在本地开发时，我们可以用Python自带的WSGI服务器，但是，在服务器上，显然不能用自带的这个开发版服务器。可以选择的WSGI服务器很多，我们选gunicorn：它用类似Nginx的Master-Worker模式，同时可以提供gevent的支持，不用修改代码，就能获得极高的性能。

此外，我们还需要一个高性能Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给gunicorn处理。gunicorn负责调用我们的Python代码，这个模型如下：



Nginx负责分发请求：



在服务器端，我们需要定义好部署的目录结构：

```
/
+- srv/
   +- awesome/       <-- Web App根目录
      +- www/        <-- 存放Python源码
      |  +- static/  <-- 存放静态资源文件
      +- log/        <-- 存放log
```

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把 `www` 作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本：

```
michael@ubuntu:/srv/awesome$ ls -l
total 32
drwxr-xr-x 2 www-data www-data 4096 Jun  5 17:38 log
lrwxrwxrwx 1 root     root       21 Jun  5 17:50 www -> www-14-06-05_17.56.16
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:26 www-14-06-05_15.26.45
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:31 www-14-06-05_15.31.03
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:32 www-14-06-05_15.32.39
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:34 www-14-06-05_17.39.28
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:35 www-14-06-05_17.41.22
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:39 www-14-06-05_17.45.57
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:50 www-14-06-05_17.56.16
michael@ubuntu:/srv/awesome$ _
```

而Nginx和gunicorn的配置文件只需要指向 `www` 目录即可。

Nginx可以作为服务进程直接启动，但gunicorn还不行，所以，Supervisor登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

- Nginx：高性能Web服务器+负责反向代理；
- gunicorn：高性能WSGI服务器；
- gevent：把Python同步代码变成异步协程的库；
- Supervisor：监控服务进程的工具；
- MySQL：数据库服务。

在Linux服务器上用apt可以直接安装上述服务：

```
$ sudo apt-get install nginx gunicorn python-gevent supervisor mysq
```

然后，再把我们自己的Web App用到的Python库安装了：

```
$ sudo apt-get install python-jinja2 python-mysql.connector
```

在服务器上创建目录 `/srv/awesome/` 以及相应的子目录。

在服务器上初始化MySQL数据库，把数据库初始化脚本 `schema.sql` 复制到服务器上执行：

```
$ mysql -u root -p < schema.sql
```

服务器端准备就绪。

部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。Fabric就是一个自动化部署工具。由于Fabric是用Python开发的，所以，部署脚本也是用Python来编写，非常方便！

要用Fabric部署，需要在本机（是开发机器，不是Linux服务器）安装Fabric：

```
$ easy_install fabric
```

Linux服务器上不需要安装Fabric，Fabric使用SSH直接登录服务器并执行部署命令。

下一步是编写部署脚本。Fabric的部署脚本叫 `fabfile.py` ，我们把它放到 `awesome-python-webapp` 的目录下，与 `www` 目录平级：

```
awesome-python-webapp/
+- fabfile.py
+- www/
+- ...
```

Fabric的脚本编写很简单，首先导入Fabric的API，设置部署时的变量：

```
# fabfile.py
import os, re
from datetime import datetime

# 导入Fabric API:
from fabric.api import *

# 服务器登录用户名:
env.user = 'michael'
# sudo用户为root:
env.sudo_user = 'root'
# 服务器地址，可以有多个，依次部署:
env.hosts = ['192.168.0.3']

# 服务器MySQL用户名和口令:
db_user = 'www-data'
db_password = 'www-data'
```

然后，每个Python函数都是一个任务。我们先编写一个打包的任务：

```
_TAR_FILE = 'dist-awesome.tar.gz'

def build():
    includes = ['static', 'templates', 'transwarp', 'favicon.ico',
    excludes = ['test', '.*', '*.pyc', '*.pyo']
    local('rm -f dist/%s' % _TAR_FILE)
    with lcd(os.path.join(os.path.abspath('.'), 'www')):
        cmd = ['tar', '--dereference', '-czvf', '../dist/%s' % _TAR
        cmd.extend(['--exclude=\'%s\'' % ex for ex in excludes])
        cmd.extend(includes)
        local(' '.join(cmd))
```

Fabric提供 `local('...')` 来运行本地命令， `with lcd(path)` 可以把当前命令的目录设定为 `lcd()` 指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要Cgywin环境。

在 `awesome-python-webapp` 目录下运行：

```
$ fab build
```

看看是否在 `dist` 目录下创建了 `dist-awesome.tar.gz` 的文件。

打包后，我们就可以继续编写 `deploy` 任务，把打包文件上传至服务器，解压，重置 `www` 软链接，重启相关服务：

```
_REMOTE_TMP_TAR = '/tmp/%s' % _TAR_FILE
_REMOTE_BASE_DIR = '/srv/awesome'

def deploy():
    newdir = 'www-%s' % datetime.now().strftime('%y-%m-%d_%H.%M.%S
    # 删除已有的tar文件:
    run('rm -f %s' % _REMOTE_TMP_TAR)
    # 上传新的tar文件:
    put('dist/%s' % _TAR_FILE, _REMOTE_TMP_TAR)
    # 创建新目录:
    with cd(_REMOTE_BASE_DIR):
        sudo('mkdir %s' % newdir)
    # 解压到新目录:
    with cd('%s/%s' % (_REMOTE_BASE_DIR, newdir)):
        sudo('tar -xzvf %s' % _REMOTE_TMP_TAR)
    # 重置软链接:
    with cd(_REMOTE_BASE_DIR):
        sudo('rm -f www')
        sudo('ln -s %s www' % newdir)
        sudo('chown www-data:www-data www')
        sudo('chown -R www-data:www-data %s' % newdir)
    # 重启Python服务和nginx服务器:
    with settings(warn_only=True):
        sudo('supervisorctl stop awesome')
        sudo('supervisorctl start awesome')
        sudo('/etc/init.d/nginx reload')
```

注意 `run()` 函数执行的命令是在服务器上运行， `with cd(path)` 和 `with lcd(path)` 类似，把当前目录在服务器端设置为 `cd()` 指定的目录。如果一个命令需要sudo权限，就不能用 `run()` ，而是用 `sudo()` 来执行。

配置Supervisor

上面让Supervisor重启gunicorn的命令会失败，因为我们还没有配置Supervisor呢。

编写一个Supervisor的配置文件 `awesome.conf` ，存放到 `/etc/supervisor/conf.d/` 目录下：

```
[program:awesome]
command     = /usr/bin/gunicorn --bind 127.0.0.1:9000 --workers 1 -
directory   = /srv/awesome/www
user        = www-data
startsecs   = 3

redirect_stderr         = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups  = 10
stdout_logfile          = /srv/awesome/log/app.log
```

配置文件通过 `[program:awesome]` 指定服务名为 `awesome` ，`command` 指定启动gunicorn的命令行，设定gunicorn的启动端口为9000，WSGI处理函数入口为 `wsgiapp:application` 。

然后重启Supervisor后，就可以随时启动和停止Supervisor管理的服务了：

```
$ sudo supervisorctl reload
$ sudo supervisorctl start awesome
$ sudo supervisorctl status
awesome                RUNNING    pid 1401, uptime 5:01:34
```

配置Nginx

Supervisor只负责运行gunicorn，我们还需要配置Nginx。把配置文件 `awesome` 放到 `/etc/nginx/sites-available/` 目录下：

```
server {
    listen      80; # 监听80端口

    root       /srv/awesome/www;
    access_log /srv/awesome/log/access_log;
    error_log  /srv/awesome/log/error_log;

    # server_name awesome.liaoxuefeng.com; # 配置域名

    # 处理静态文件/favicon.ico:
    location /favicon.ico {
        root /srv/awesome/www;
    }

    # 处理静态资源:
    location ~ ^\/static\/.*$ {
        root /srv/awesome/www;
    }

    # 动态请求转发到9000端口(gunicorn):
    location / {
        proxy_pass       http://127.0.0.1:9000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_fo
    }
}
```

然后在 `/etc/nginx/sites-enabled/` 目录下创建软链接：

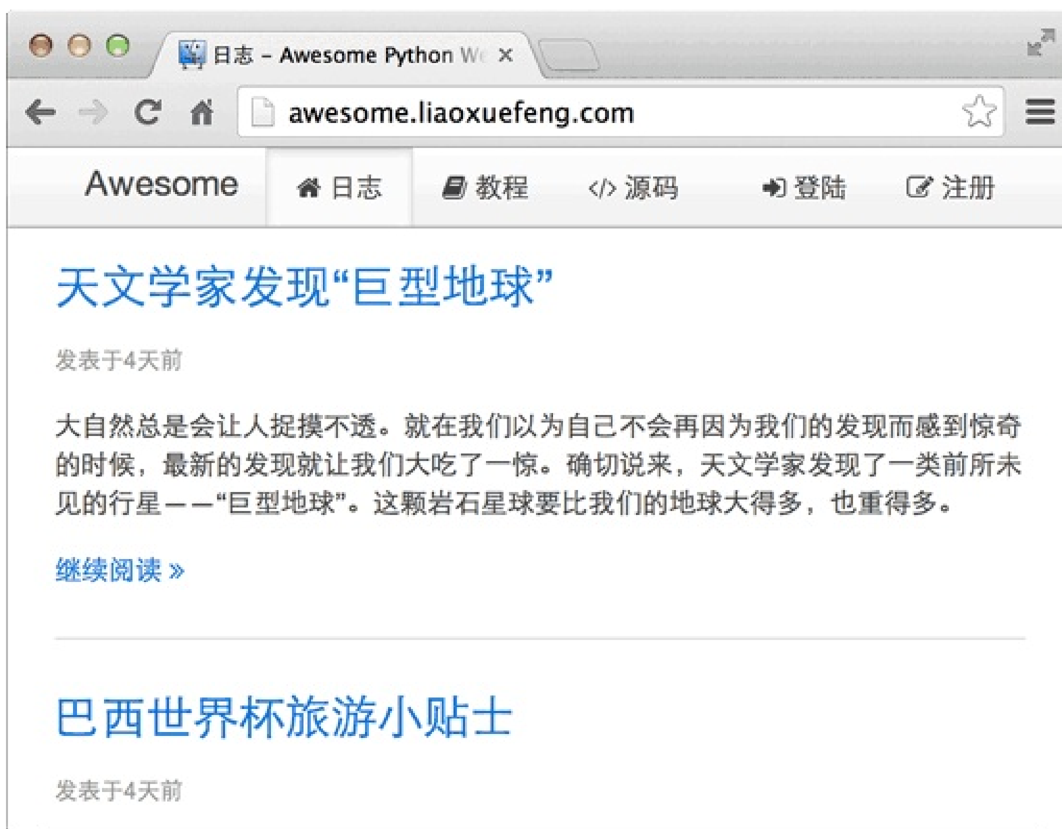
```
$ pwd
/etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/awesome .
```

让Nginx重新加载配置文件，不出意外，我们的 `awesome-python-webapp` 应该正常运行：

```
$ sudo /etc/init.d/nginx reload
```

如果有任何错误，都可以在 `/srv/awesome/log` 下查找Nginx和App本身的log。如果Supervisor启动时报错，可以在 `/var/log/supervisor` 下查看Supervisor的log。

如果一切顺利，你可以在浏览器中访问Linux服务器上的 `awesome-python-webapp` 了：



如果在开发环境更新了代码，只需要在命令行执行：

```
$ fab build
$ fab deploy
```

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

友情链接

嫌国外网速慢的童鞋请移步网易和搜狐的镜像站点：

http://mirrors.163.com/

http://mirrors.sohu.com/

Day 16 - 编写移动App

网站部署上线后，还缺点啥呢？

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动App，出门根本不好意思跟人打招呼。

所以， `awesome-python-webapp` 必须得有一个移动App版本！

开发iPhone版本

我们首先来看看如何开发iPhone App。前置条件：一台Mac电脑，安装XCode和最新的iOS SDK。

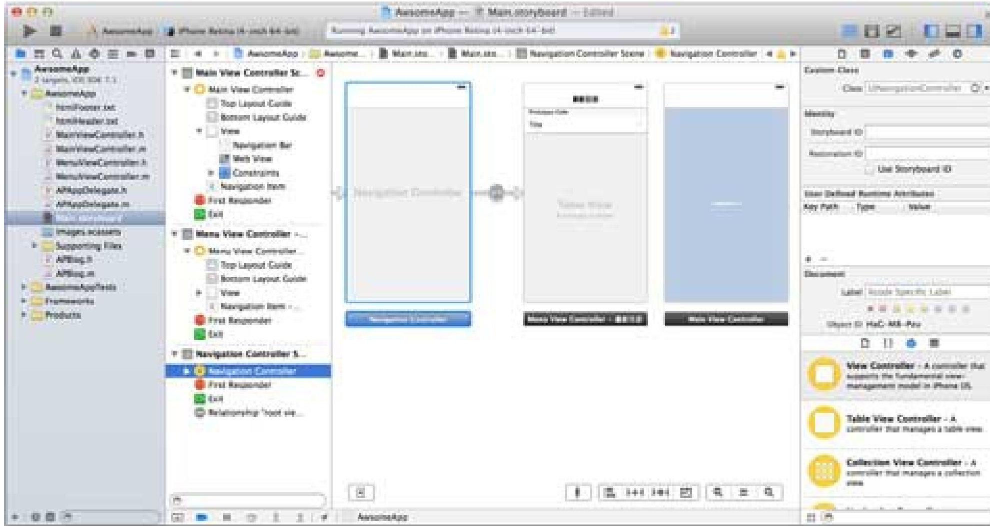
在使用MVVM编写前端页面时，我们就能感受到，用REST API封装网站后台的功能，不但能清晰地分离前端页面和后台逻辑，现在这个好处更加明显，移动App也可以通过REST API从后端拿到数据。

我们来设计一个简化版的iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容：



只需要调用API： `/api/blogs` 。

在XCode中完成App编写：



由于我们的教程是Python，关于如何开发iOS，请移步Develop Apps for iOS。

点击下载iOS App源码。

如何编写Android App？这个当成作业了。

期末总结

终于到了期末总结的时刻了！

经过一段时间的学习，相信你对Python已经初步掌握。一开始，可能觉得Python上手很容易，可是越往后学，会越困难，有的时候，发现理解不了代码，这时，不妨停下来思考一下，先把概念搞清楚，代码自然就明白了。

Python非常适合初学者用来进入计算机编程领域。Python属于非常高级的语言，掌握了这门高级语言，就对计算机编程的核心思想——抽象有了初步理解。如果希望继续深入学习计算机编程，可以学习C、JavaScript、Lisp等不同类型的语言，只有多掌握不同领域的语言，有比较才更有收获。

谢谢学习！

Python3教程

这是小白的Python新手教程，具有如下特点：

中文，免费，零起点，完整示例，基于最新的**Python 3**版本。

Python是一种计算机程序设计语言。你可能已经听说过很多种流行的编程语言，比如非常难学的C语言，非常流行的Java语言，适合初学者的Basic语言，适合网页编程的JavaScript语言等等。

那Python是一种什么语言？

首先，我们普及一下编程语言的基础知识。用任何编程语言来开发程序，都是为了让计算机干活，比如下载一个MP3，编写一个文档等等，而计算机干活的CPU只认识机器指令，所以，尽管不同的编程语言差异极大，最后都得“翻译”成CPU可以执行的机器指令。而不同的编程语言，干同一个活，编写的代码量，差距也很大。

比如，完成同一个任务，C语言要写1000行代码，Java只需要写100行，而Python可能只要20行。

所以Python是一种相当高级的语言。

你也许会问，代码少还不好？代码少的代价是运行速度慢，C程序运行1秒钟，Java程序可能需要2秒，而Python程序可能就需要10秒。

那是不是越低级的程序越难学，越高级的程序越简单？表面上来说，是的，但是，在非常高的抽象计算中，高级的Python程序设计也是非常难学的，所以，高级程序语言不等于简单。

但是，对于初学者和完成普通任务，Python语言是非常简单易用的。连Google都在大规模使用Python，你就不用担心学了会没用。

用Python可以做什么？可以做日常任务，比如自动备份你的MP3；可以做网站，很多著名的网站包括YouTube就是Python写的；可以做网络游戏的后台，很多在线游戏的后台都是Python开发的。总之就是能干很多很多事啦。

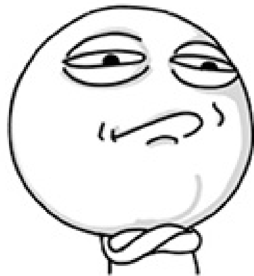
Python当然也有不能干的事情，比如写操作系统，这个只能用C语言写；写手机应用，只能用Swift/Objective-C（针对iPhone）和Java（针对Android）；写3D游戏，最好用C或C++。

如果你是小白用户，满足以下条件：

- 会使用电脑，但从来没写过程序；
- 还记得初中数学学的方程式和一点点代数知识；
- 想从编程小白变成专业的软件架构师；
- 每天能抽出半个小时学习。

不要再犹豫了，这个教程就是为你准备的！

准备好了吗？



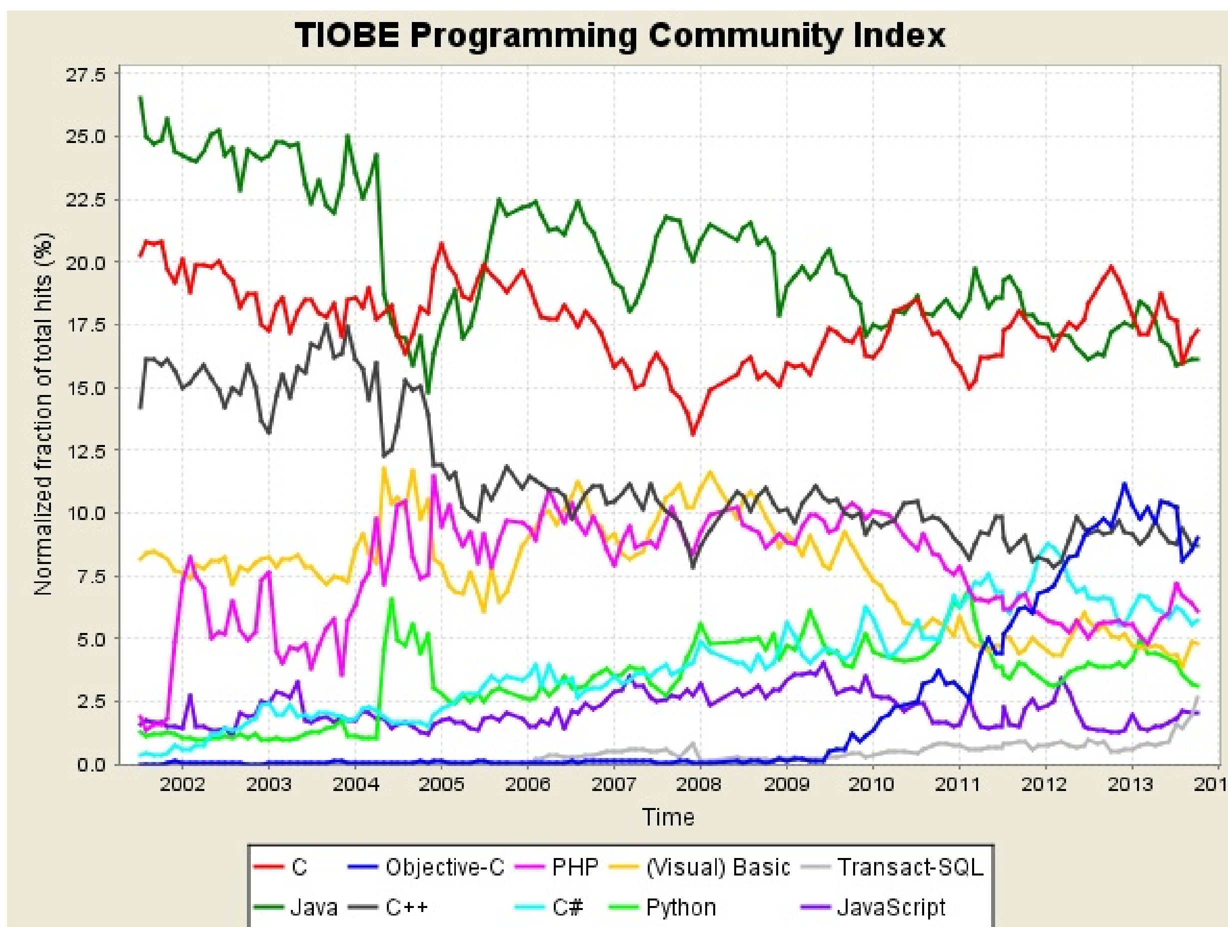
关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Scheme/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub，欢迎微博交流：@廖雪峰。

Python简介

Python是著名的"龟叔"Guido van Rossum在1989年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。

现在，全世界差不多有600多种编程语言，但流行的编程语言也就那么20来种。如果你听说过TIOBE排行榜，你就能知道编程语言的大致流行程度。这是最近10年最常用的10种编程语言的变化图：



总的来说，这几种编程语言各有千秋。C语言是可以用来编写操作系统的贴近硬件的语言，所以，C语言适合开发那些追求运行速度、充分发挥硬件性能的程序。而Python是用来编写应用程序的高级编程语言。

当你用一种语言开始作真正的软件开发时，你除了编写代码外，还需要很多基本的已经写好的现成的东西，来帮助你加快开发进度。比如说，要编写一个电子邮件客户端，如果先从最底层开始编写网络协议相关的代码，那估计一年半载也开发不出

来。高级编程语言通常都会提供一个比较完善的基础代码库，让你能直接调用，比如，针对电子邮件协议的SMTP库，针对桌面环境的GUI库，在这些已有的代码库的基础上开发，一个电子邮件客户端几天就能开发出来。

Python就为我们提供了非常完善的基础代码库，覆盖了网络、文件、GUI、数据库、文本等大量内容，被形象地称作“内置电池（batteries included）”。用Python开发，许多功能不必从零编写，直接使用现成的即可。

除了内置的库外，Python还有大量的第三方库，也就是别人开发的，供你直接使用的东西。当然，如果你开发的代码通过很好的封装，也可以作为第三方库给别人使用。

许多大型网站就是用Python开发的，例如YouTube、Instagram，还有国内的豆瓣。很多大公司，包括Google、Yahoo等，甚至NASA（美国航空航天局）都大量地使用Python。

龟叔给Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总的来说，Python的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那Python适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说Python的缺点。

任何编程语言都有缺点，Python也不例外。优点说过了，那Python有哪些缺点呢？

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

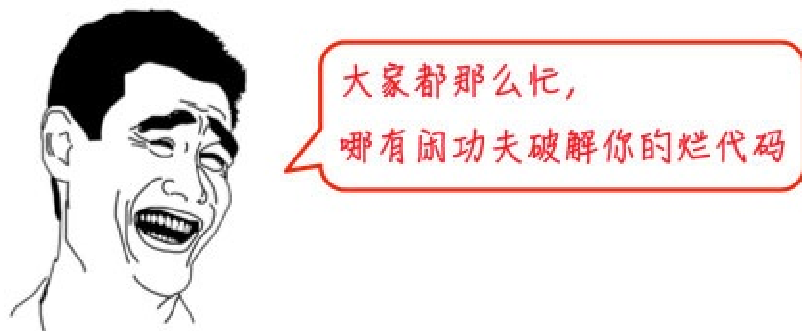
但是大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载MP3的网络应用程序，C程序的运行时间需要0.001秒，而Python程序的运行时间需要0.1秒，慢了100倍，但由于网络更慢，需要等待1秒，你想，用户能感觉到1.001秒和1.1秒的区别吗？这就好比F1赛车和普通的出租车在北京三环路上行驶的道理一样，虽然F1赛车理论时速高达400公里，但由于三环路堵车的时速只有20公里，因此，作为乘客，你感觉的时速永远是20公里。



第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

这个缺点仅限于你要编写的软件需要卖给别人挣钱的时候。好消息是目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站和移动应用卖服务的模式越来越多了，后一种模式不需要把源码给别人。

再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux一样的开源代码，我们千万不要高估自己写的代码真的有非常大的“商业价值”。那些大公司的代码不愿意开放的更重要的原因是代码写得太烂了，一旦开源，就没人敢用他们的产品了。



当然，Python还有其他若干小缺点，请自行忽略，就不一一列举了。

安装Python

因为Python是跨平台的，它可以运行在Windows、Mac和各种Linux/Unix系统上。在Windows上写Python程序，放到Linux上也是能够运行的。

要开始学习Python编程，首先就得把Python安装到你的电脑里。安装后，你会得到Python解释器（就是负责运行Python程序的），一个命令行交互环境，还有一个简单的集成开发环境。

安装Python 3.5

目前，Python有两个版本，一个是2.x版，一个是3.x版，这两个版本是不兼容的。由于3.x版越来越普及，我们的教程将以最新的Python 3.5版本为基础。请确保你的电脑上安装的Python版本是最新的3.5.x，这样，你才能无痛学习这个教程。

在Mac上安装Python

如果你正在使用Mac，系统是OS X 10.8~10.10，那么系统自带的Python版本是2.7。要安装最新的Python 3.5，有两个方法：

方法一：从Python官网下载Python 3.5的安装程序（网速慢的同学请移步国内镜像），双击运行并安装；

方法二：如果安装了Homebrew，直接通过命令 `brew install python3` 安装即可。

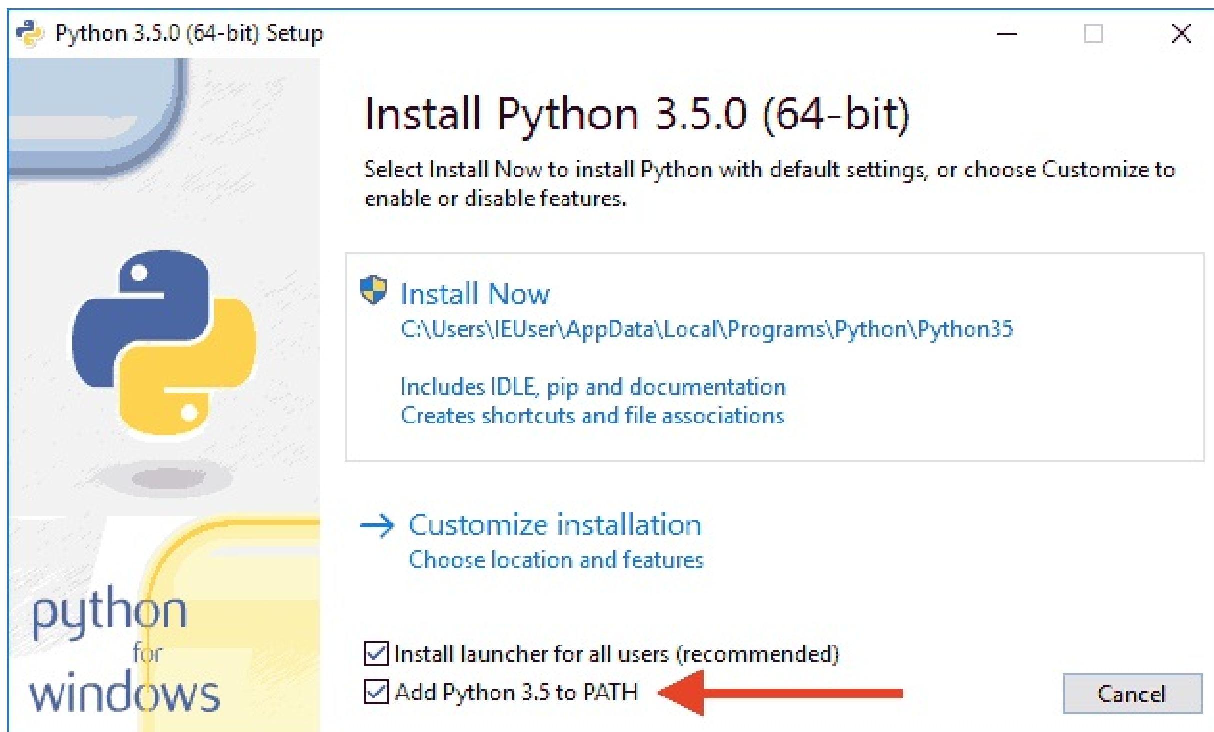
在Linux上安装Python

如果你正在使用Linux，那我可以假定你有Linux系统管理经验，自行安装Python 3应该没有问题，否则，请换回Windows系统。

对于大量的目前仍在使用Windows的同学，如果短期内没有打算换Mac，就可以继续阅读以下内容。

在Windows上安装Python

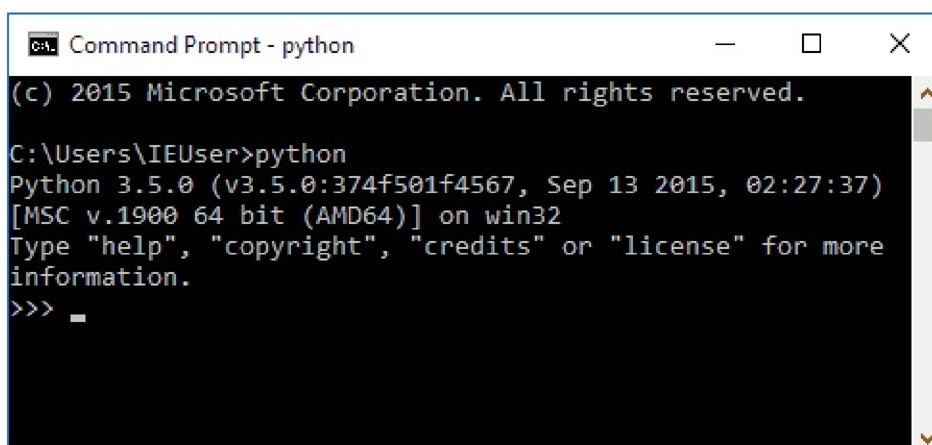
首先，根据你的Windows版本（64位还是32位）从Python的官方网站下载Python 3.5对应的64位安装程序或32位安装程序（网速慢的同学请移步国内镜像），然后，运行下载的EXE安装包：



特别要注意勾上`Add Python 3.5 to PATH`，然后点“Install Now”即可完成安装。

默认会安装到`C:\Python35`目录下，然后打开命令提示符窗口，敲入python后，会出现两种情况：

情况一：



看到上面的画面，就说明Python安装成功！

你看到提示符 `>>>` 就表示我们已经在Python交互式环境中了，可以输入任何Python代码，回车后会立刻得到执行结果。现在，输入 `exit()` 并回车，就可以退出Python交互式环境（直接关掉命令行窗口也可以）。

情况二：得到一个错误：

```
‘python’ 不是内部或外部命令，也不是可运行的程序或批处理文件。
```

```
Command Prompt

C:\Users\IEUser>python
'python' is not recognized as an internal or external comm
and,
operable program or batch file.

C:\Users\IEUser>_
```

这是因为Windows会根据一个 `Path` 的环境变量设定的路径去查找 `python.exe` ，如果没找到，就会报错。如果在安装时漏掉了勾选 `Add Python 3.5 to PATH` ，那就要手动把 `python.exe` 所在的路径添加到Path中。

如果你不知道怎么修改环境变量，建议把Python安装程序重新运行一遍，务必记得勾上 `Add Python 3.5 to PATH` 。

小结

学会如何把Python安装到计算机中，并且熟练打开和退出Python交互式环境。

在Windows上运行Python时，请先启动命令行，然后运行 `python` 。

在Mac和Linux上运行Python时，请打开终端，然后运行 `python3` 。

Python解释器

当我们编写Python代码时，我们得到的是一个包含Python代码的以 `.py` 为扩展名的文本文件。要运行代码，就需要Python解释器去执行 `.py` 文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

CPython

当我们从Python官方网站下载并安装好Python 3.5后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行 `python` 就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用 `>>>` 作为提示符，而IPython用 `In [序号]:` 作为提示符。

PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用JIT技术，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解PyPy和CPython的不同点。

Jython

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

IronPython

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

小结

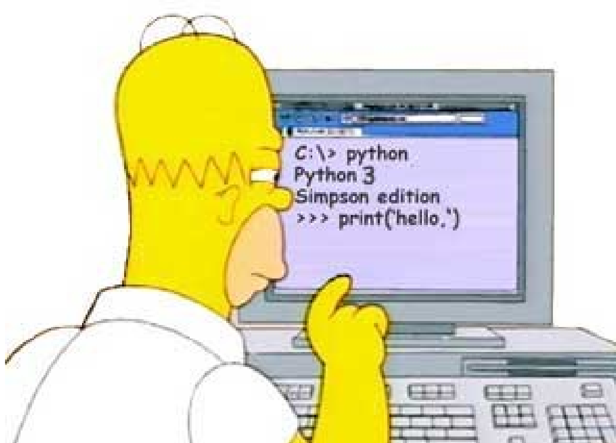
Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 3.5版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

第一个Python程序

现在，了解了如何启动和退出Python的交互式环境，我们就可以正式开始编写Python代码了。

在写代码之前，请千万不要用“复制”-“粘贴”把代码从页面粘贴到你自己的电脑上。写程序也讲究一个感觉，你需要一个字母一个字母地把代码自己敲进去，在敲代码的过程中，初学者经常会敲错代码，所以，你需要仔细地检查、对照，才能以最快的速度掌握如何写程序。



在交互式环境的提示符 `>>>` 下，直接输入代码，按回车，就可以立刻得到代码执行结果。现在，试试输入 `100+200` ，看看计算结果是不是300：

```
>>> 100+200
300
```

很简单吧，任何有效的数学计算都可以算出来。

如果要让Python打印出指定的文字，可以用 `print()` 函数，然后把希望打印的文字用单引号或者双引号括起来，但不能混用单引号和双引号：

```
>>> print('hello, world')
hello, world
```

这种用单引号或者双引号括起来的文本在程序中叫字符串，今后我们还会经常遇到。

最后，用 `exit()` 退出Python，我们的第一个Python程序完成！唯一的缺憾是没有保存下来，下次运行时还要再输入一遍代码。

小结

在Python交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。

使用文本编辑器

在Python的交互式命令行写程序，好处是一下就能得到结果，坏处是没法保存，下次还想运行的时候，还得再敲一遍。

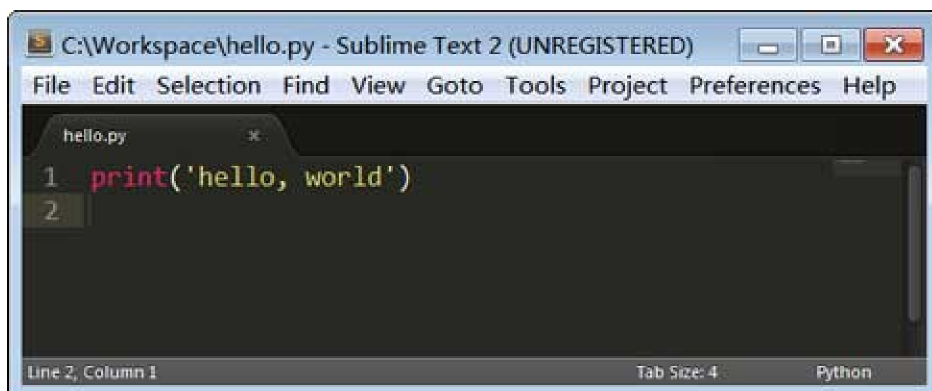
所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

现在，我们就把上次的 `'hello, world'` 程序用文本编辑器写出来，保存下来。

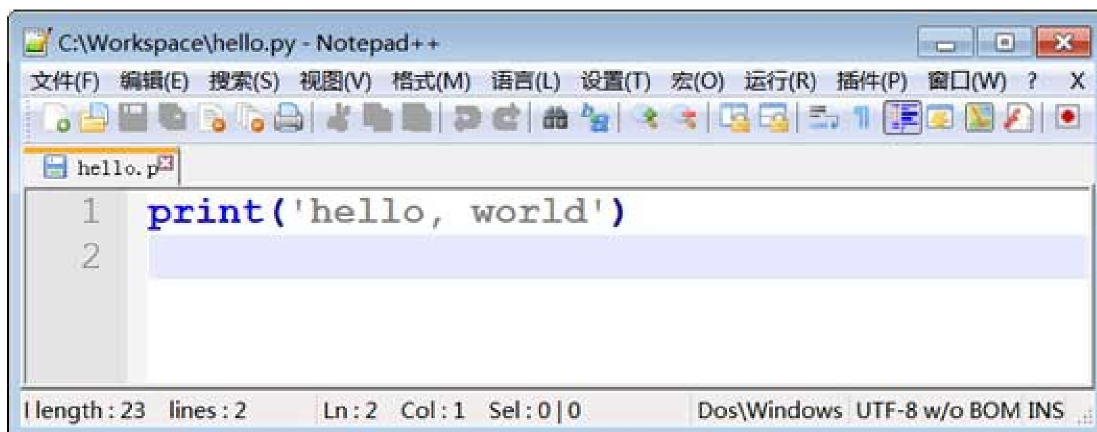
那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是Sublime Text，免费使用，但是不付费会弹出提示框：



一个是Notepad++，免费使用，有中文界面：



请注意，用哪个都行，但是绝对不能用Word和Windows自带的记事本。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

```
print('hello, world')
```

注意 `print` 前面不要有任何空格。然后，选择一个目录，例如 `C:\work` ，把文件保存为 `hello.py` ，就可以打开命令行窗口，把当前目录切换到 `hello.py` 所在目录，就可以运行这个程序了：

```
C:\work>python hello.py
hello, world
```

也可以保存为别的名字，比如 `first.py` ，但是必须要以 `.py` 结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有 `hello.py` 这个文件，运行 `python hello.py` 就会报错：

```
C:\Users\IEUser>python hello.py
python: can't open file 'hello.py': [Errno 2] No such file or direc
```

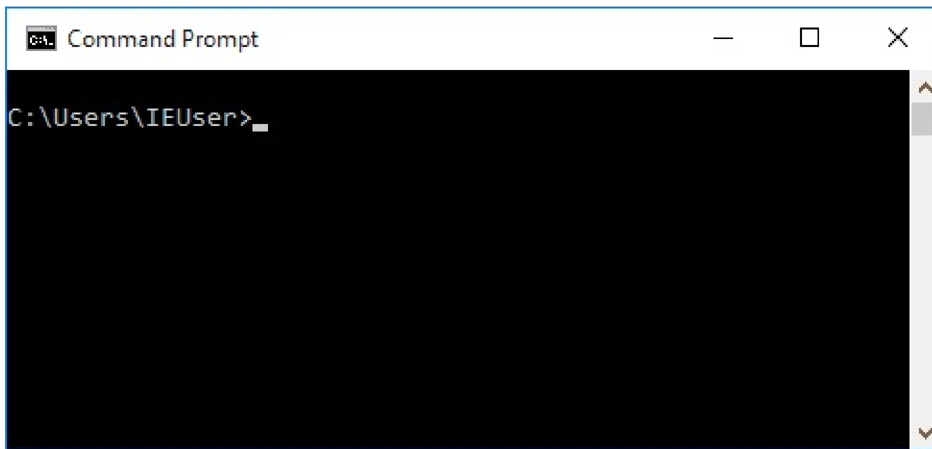
报错的意思就是，无法打开 `hello.py` 这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。如果 `hello.py` 存放在另外一个目录下，要首先用 `cd` 命令切换当前目录：

http://michaelliao.gitcafe.io/video/py/run-py3-hello.mp4

命令行模式和Python交互模式

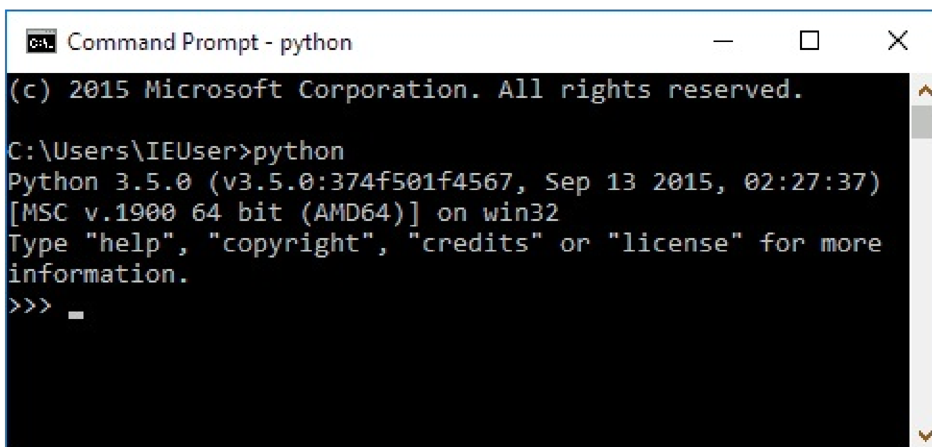
请注意区分命令行模式和Python交互模式。

看到类似 `C:\>` 是在Windows提供的命令行模式：



在命令行模式下，可以执行 `python` 进入Python交互式环境，也可以执行 `python hello.py` 运行一个 `.py` 文件。

看到 `>>>` 是在Python交互式环境下：



在Python交互式环境下，只能输入Python代码并立刻执行。

此外，在命令行模式运行 `.py` 文件和在Python交互式环境下直接运行Python代码有所不同。Python交互式环境会把每一行Python代码的结果自动打印出来，但是，直接运行Python代码却不会。

例如，在Python交互式环境下，输入：

```
>>> 100 + 200 + 300
600
```

直接可以看到结果 `600` 。

但是，写一个 `calc.py` 的文件，内容如下：

```
100 + 200 + 300
```

然后在命令行模式下执行：

```
C:\work>python calc.py
```

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用 `print()` 打印出来。把 `calc.py` 改造一下：

```
print(100 + 200 + 300)
```

再执行，就可以看到结果：

```
C:\work>python calc.py
600
```

直接运行py文件

还有同学问，能不能像.exe文件那样直接运行 `.py` 文件呢？在Windows上是不行的，但是，在Mac和Linux上是可以的，方法是在 `.py` 文件的第一行加上一个特殊的注释：

```
#!/usr/bin/env python3

print('hello, world')
```

然后，通过命令给 `hello.py` 以执行权限：

```
$ chmod a+x hello.py
```

就可以直接运行 `hello.py` 了，比如在Mac下运行：

```
michael — bash — 80×24
Last login: Sun Dec 15 18:44:13 on ttys000
Mavericks:~ michael$ ./hello.py
hello, world
Mavericks:~ michael$
```

小结

用文本编辑器写Python程序，然后保存为后缀为 `.py` 的文件，就可以用Python直接运行这个程序了。

Python的交互模式和直接运行 `.py` 文件有什么区别呢？

直接输入 `python` 进入交互模式，相当于启动了Python解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行 `.py` 文件相当于启动了Python解释器，然后一次性把 `.py` 文件的源代码给执行了，你是没有机会以交互的方式输入源代码的。

用Python开发程序，完全可以一边在文本编辑器里写代码，一边开一个交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

参考源码

hello.py

Python代码运行助手

Python代码运行助手可以让你在线输入Python代码，然后通过本机运行的一个Python脚本来执行代码。原理如下：

- 在网页输入代码：

```
# -*- coding: utf-8 -*-

def normalize(name):
    return name.capitalize()

# 测试:
L1 = ['adam', 'LISA', 'barT']
L2 = list(map(normalize, L1))
print(L2)
```

▶ Run

- 点击 `Run` 按钮，代码被发送到本机正在运行的Python代码运行助手；
- Python代码运行助手将代码保存为临时文件，然后调用Python解释器执行代码；
- 网页显示代码执行结果：



下载

点击右键，目标另存为：learning.py

备用下载地址：learning.py

运行

在存放 `learning.py` 的目录下运行命令：

```
C:\Users\michael\Downloads> python learning.py
```

如果看到 `Ready for Python code on port 39093...` 表示运行成功，不要关闭命令行窗口，最小化放到后台运行即可：



试试效果

需要支持HTML5的浏览器：

- IE >= 9
- Firefox
- Chrome
- Sarafi

```
# 测试代码:

print('Hello, world')
```

输入和输出

输出

用 `print()` 在括号中加上字符串，就可以向屏幕上输出指定的文字。比如输出 `'hello, world'` ，用代码实现如下：

```
>>> print('hello, world')
```

`print()` 函数也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
>>> print('The quick brown fox', 'jumps over', 'the lazy dog')
The quick brown fox jumps over the lazy dog
```

`print()` 会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：

```
print('The quick brown fox' , 'jumps over' , 'the lazy dog')
The quick brown fox jumps over the lazy dog
```

`print()` 也可以打印整数，或者计算结果：

```
>>> print(300)
300
>>> print(100 + 200)
300
```

因此，我们可以把计算 `100 + 200` 的结果打印得更漂亮一点：

```
>>> print('100 + 200 =', 100 + 200)
100 + 200 = 300
```

注意，对于`100 + 200`，Python解释器自动计算出结果`300`，但是，`'100 + 200 ='`是字符串而非数学公式，Python把它视为字符串，请自行解释上述打印结果。

输入

现在，你已经可以用`print()`输出你想要的结果了。但是，如果要让用户从电脑输入一些字符怎么办？Python提供了一个`input()`，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

```
>>> name = input()
Michael
```

当你输入`name = input()`并按下回车后，Python交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

输入完成后，不会有任何提示，Python交互式命令行又回到`>>>`状态了。那我们刚才输入的内容到哪去了？答案是存放到`name`变量里了。可以直接输入`name`查看变量内容：

```
>>> name
'Michael'
```

什么是变量？请回忆初中数学所学的代数基础知识：

设正方形的边长为`a`，则正方形的面积为`a x a`。把边长`a`看做一个变量，我们就可以根据`a`的值计算正方形的面积，比如：

若a=2，则面积为a x a = 2 x 2 = 4；

若a=3.5，则面积为a x a = 3.5 x 3.5 = 12.25。

在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此，`name`作为一个变量就是一个字符串。

要打印出`name`变量的内容，除了直接写`name`然后按回车外，还可以用`print()`函数：

```
>>> print(name)
Michael
```

有了输入和输出，我们就可以把上次打印 `'hello, world'` 的程序改成有点意义的程序了：

```
name = input()
print('hello,', name)
```

运行上面的程序，第一行代码会让用户输入任意字符作为自己的名字，然后存入 `name` 变量中；第二行代码会根据用户的名字向用户说 `hello`，比如输入 `Michael`：

```
C:\Workspace> python hello.py
Michael
hello, Michael
```

但是程序运行的时候，没有任何提示信息告诉用户："嘿，赶紧输入你的名字"，这样显得很不友好。幸好，`input()` 可以让你显示一个字符串来提示用户，于是我们把代码改成：

```
name = input('please enter your name: ')
print('hello,', name)
```

再次运行这个程序，你会发现，程序一运行，会首先打印出 `please enter your name:`，这样，用户就可以根据提示，输入名字后，得到 `hello, xxx` 的输出：

```
C:\Workspace> python hello.py
please enter your name: Michael
hello, Michael
```

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

小结

任何计算机程序都是为了执行一个特定的任务，有了输入，用户才能告诉计算机程序所需的信息，有了输出，程序运行后才能告诉用户任务的结果。

输入是Input，输出是Output，因此，我们把输入输出统称为Input/Output，或者简写为IO。

`input()` 和 `print()` 是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

练习

请利用 `print()` 输出 `1024 * 768 = xxx` ：

```
# -*- coding: utf-8 -*-

print(???)
```

参考源码

do_input.py

Python基础

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# print absolute value of an integer:
a = 100
if a >= 0:
    print(a)
else:
    print(-a)
```

以 `#` 开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号 `:` 结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制－粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

小结

Python使用缩进来组织代码块，请务必遵守约定俗成的习惯，坚持使用4个空格的缩进。

在文本编辑器中，需要设置把Tab自动转换为4个空格，确保不混用Tab和空格。

数据类型和变量

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

整数

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：`1`，`100`，`-8080`，`0`，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用`0x`前缀和0-9，a-f表示，例如：`0xff00`，`0xa5b4c3d2`，等等。

浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，1.23x10⁹和12.3x10⁸是完全相等的。浮点数可以用数学写法，如`1.23`，`3.14`，`-9.01`，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x10⁹就是`1.23e9`，或者`12.3e8`，0.000012可以写成`1.2e-5`，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

字符串

字符串是以单引号`'`或双引号`"`括起来的任意文本，比如`'abc'`，`"xyz"`等等。请注意，`''`或`""`本身只是一种表示方式，不是字符串的一部分，因此，字符串`'abc'`只有`a`，`b`，`c`这3个字符。如果`'`本身也是一个字符，那就可

以用 `""` 括起来，比如 `"I'm OK"` 包含的字符是 `I` ， `'` ， `m` ，空格， `O` ， `K` 这6个字符。

如果字符串内部既包含 `'` 又包含 `"` 怎么办？可以用转义字符 `\` 来标识，比如：

```
'I\'m \"OK\"!'
```

表示的字符串内容是：

```
I'm "OK"!
```

转义字符 `\` 可以转义很多字符，比如 `\n` 表示换行， `\t` 表示制表符，字符 `\` 本身也要转义，所以 `\\` 表示的字符就是 `\` ，可以在Python的交互式命令行用 `print()` 打印字符串看看：

```
>>> print('I\'m ok.')
I'm ok.
>>> print('I\'m learning\nPython.')
I'm learning
Python.
>>> print('\\\n\\')
\
\
```

如果字符串里面有很多字符都需要转义，就需要加很多 `\` ，为了简化，Python还允许用 `r''` 表示 `''` 内部的字符串默认不转义，可以自己试试：

```
>>> print('\\\t\\')
\       \
>>> print(r'\\\t\\')
\\\t\\
```

如果字符串内部有很多换行，用 `\n` 写在一行里不好阅读，为了简化，Python允许用 `'''...'''` 的格式表示多行内容，可以自己试试：

```
>>> print('''line1
... line2
... line3''')
line1
line2
line3
```

上面是在交互式命令行内输入，注意在输入多行内容时，提示符由 `>>>` 变为 `...` ，提示你可以接着上一行输入。如果写成程序，就是：

```
print('''line1
line2
line3''')
```

多行字符串 `'''...'''` 还可以在前面加上 `r` 使用，请自行测试。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有 `True` 、 `False` 两种值，要么是 `True` ，要么是 `False` ，在Python中，可以直接用 `True` 、 `False` 表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用 `and` 、 `or` 和 `not` 运算。

`and` 运算是与运算，只有所有都为 `True` ， `and` 运算结果才是 `True` ：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
```

`or` 运算是或运算，只要其中有一个为 `True` ， `or` 运算结果就是 `True` ：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
```

`not` 运算是非运算，它是一个单目运算符，把 `True` 变成 `False` ， `False` 变成 `True` ：

```
>>> not True
False
>>> not False
True
>>> not 1 > 2
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:
    print('adult')
else:
    print('teenager')
```

空值

空值是Python里一个特殊的值，用 `None` 表示。 `None` 不能理解为 `0` ，因为 `0` 是有意义的，而 `None` 是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和 `_` 的组合，且不能用数字开头，比如：

```
a = 1
```

变量 `a` 是一个整数。

```
t_007 = 'T007'
```

变量 `t_007` 是一个字符串。

```
Answer = True
```

变量 `Answer` 是一个布尔值 `True` 。

在Python中，等号 `=` 是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

```
a = 123 # a是整数
print(a)
a = 'ABC' # a变为字符串
print(a)
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
x = 10
x = x + 2
```

如果从数学上理解 `x = x + 2` 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 `x + 2` ，得到结果 `12` ，再赋给变量 `x` 。由于 `x` 之前的值是 `10` ，重新赋值后， `x` 的值变成 `12` 。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python解释器干了两件事情：

1. 在内存中创建了一个 `'ABC'` 的字符串；
2. 在内存中创建了一个名为 `a` 的变量，并把它指向 `'ABC'` 。

也可以把一个变量 `a` 赋值给另一个变量 `b` ，这个操作实际上是把变量 `b` 指向变量 `a` 所指向的数据，例如下面的代码：

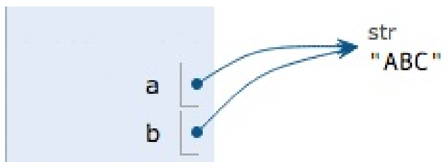
```
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

最后一行打印出变量 `b` 的内容到底是 `'ABC'` 呢还是 `'XYZ'` ？如果从数学意义上理解，就会错误地得出 `b` 和 `a` 相同，也应该是 `'XYZ'` ，但实际上 `b` 的值是 `'ABC'` ，让我们一行一行地执行代码，就可以看到到底发生了什么事：

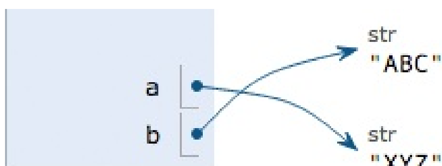
执行 `a = 'ABC'` ，解释器创建了字符串 `'ABC'` 和变量 `a` ，并把 `a` 指向 `'ABC'` ：



执行 `b = a` ，解释器创建了变量 `b` ，并把 `b` 指向 `a` 指向的字符串 `'ABC'` ：



执行 `a = 'XYZ'` ，解释器创建了字符串'XYZ'，并把 `a` 的指向改为 `'XYZ'` ，但 `b` 并没有更改：



所以，最后打印变量 `b` 的结果自然是 `'ABC'` 了。

常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

```
PI = 3.14159265359
```

但事实上 `PI` 仍然是一个变量，Python根本没有任何机制保证 `PI` 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 `PI` 的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是 `/` ：

```
>>> 10 / 3
3.3333333333333335
```

`/` 除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

```
>>> 9 / 3
3.0
```

还有一种除法是 `//` ，称为地板除，两个整数的除法仍然是整数：

```
>>> 10 // 3
3
```

你没有看错，整数的地板除 `//` 永远是整数，即使除不尽。要做精确的除法，使用 `/` 就可以。

因为 `//` 除法只取结果的整数部分，所以Python还提供一个余数运算，可以得到两个整数相除的余数：

```
>>> 10 % 3
1
```

无论整数做 `//` 除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

练习

请打印出以下变量的值：

```
n = 123
f = 456.789
s1 = 'Hello, world'
s2 = 'Hello, \'Adam\''
s3 = r'Hello, "Bart"'
s4 = r'''Hello,
Lisa!'''
```

小结

Python支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在 `-2147483648` - `2147483647` 。

Python的浮点数也没有大小限制，但是超出一定范围就直接表示为 `inf` （无限大）。

字符串和编码

字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是 `65535` ，4个字节可以表示的最大整数是 `4294967295` 。

由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 `ASCII` 编码，比如大写字母 `A` 的编码是 `65` ，小写字母 `z` 的编码是 `122` 。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了 `GB2312` 编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到 `Shift_JIS` 里，韩国把韩文编到 `Euc-kr` 里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。



因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母 `A` 用ASCII编码是十进制的 `65` ，二进制的 `01000001` ；

字符 `0` 用ASCII编码是十进制的 `48` ，二进制的 `00110000` ，注意字符 `'0'` 和整数 `0` 是不同的；

汉字 `中` 已经超出了ASCII编码的范围，用Unicode编码是十进制的 `20013` ，二进制的 `01001110 00101101` 。

你可以猜测，如果把ASCII编码的 `A` 用Unicode编码，只需要在前面补0就可以，因此， `A` 的Unicode编码是 `00000000 01000001` 。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的 `UTF-8` 编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

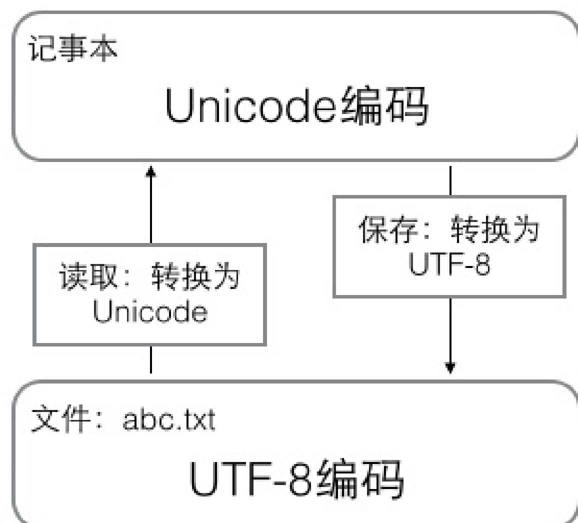
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

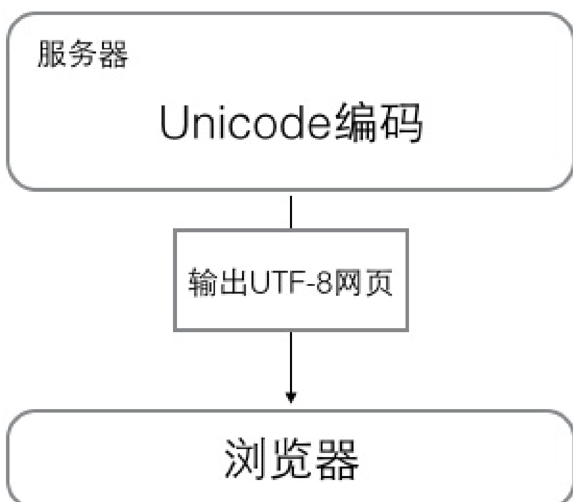
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看到很多网页的源码上会有类似 `<meta charset="UTF-8" />` 的信息，表示该网页正是用的UTF-8编码。

Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python的字符串。

在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

```
>>> print('包含中文的str')
包含中文的str
```

对于单个字符的编码，Python提供了 `ord()` 函数获取字符的整数表示， `chr()` 函数把编码转换为对应的字符：

```
>>> ord('A')
65
>>> ord('中')
20013
>>> chr(66)
'B'
>>> chr(25991)
'文'
```

如果知道字符的整数编码，还可以用十六进制这么写 `str` ：

```
>>> '\u4e2d\u6587'
'中文'
```

两种写法完全是等价的。

由于Python的字符串类型是 `str` ，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把 `str` 变为以字节为单位的 `bytes` 。

Python对 `bytes` 类型的数据用带 `b` 前缀的单引号或双引号表示：

```
x = b'ABC'
```

要注意区分 `'ABC'` 和 `b'ABC'` ，前者是 `str` ，后者虽然内容显示得和前者一样，但 `bytes` 的每个字符都只占用一个字节。

以Unicode表示的 `str` 通过 `encode()` 方法可以编码为指定的 `bytes` ，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
b'\xe4\xb8\xad\xe6\x96\x87'
>>> '中文'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in posit
```

纯英文的 `str` 可以用 `ASCII` 编码为 `bytes` ，内容是一样的，含有中文的 `str` 可以用 `UTF-8` 编码为 `bytes` 。含有中文的 `str` 无法用 `ASCII` 编码，因为中文编码的范围超过了 `ASCII` 编码的范围，Python会报错。

在 `bytes` 中，无法显示为ASCII字符的字节，用 `\x##` 显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是 `bytes` 。要把 `bytes` 变为 `str` ，就需要用 `decode()` 方法：

```
>>> b'ABC'.decode('ascii')
'ABC'
>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
```

要计算 `str` 包含多少个字符，可以用 `len()` 函数：

```
>>> len('ABC')
3
>>> len('中文')
2
```

`len()` 函数计算的是 `str` 的字符数，如果换成 `bytes` ， `len()` 函数就计算字节数：

```
>>> len(b'ABC')
3
>>> len(b'\xe4\xb8\xad\xe6\x96\x87')
6
>>> len('中文'.encode('utf-8'))
6
```

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字符只占用1个字节。

在操作字符串时，我们经常遇到 `str` 和 `bytes` 的互相转换。为了避免乱码问题，应当始终坚持使用UTF-8编码对 `str` 和 `bytes` 进行转换。

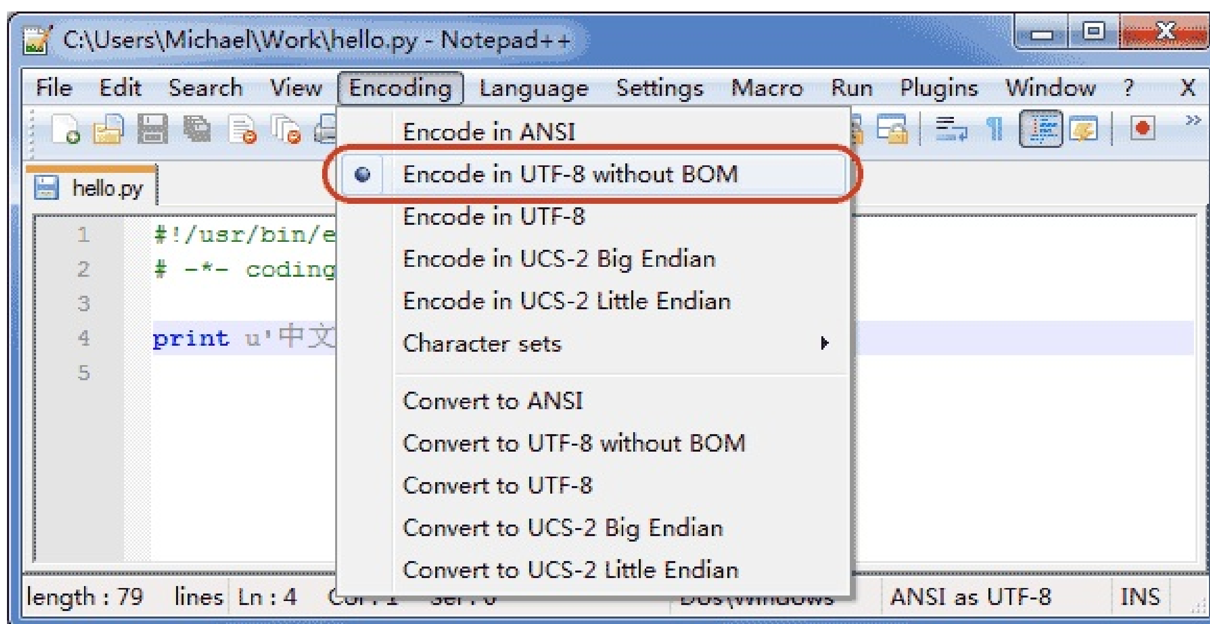
由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

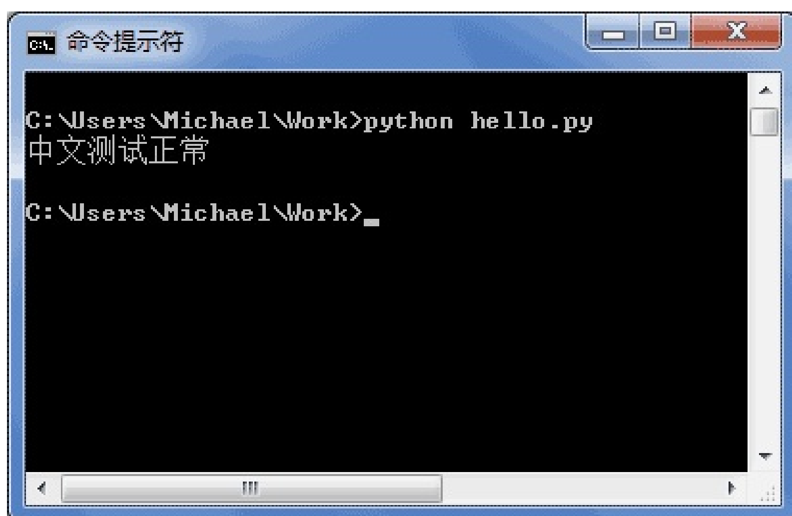
第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了UTF-8编码并不意味着你的 `.py` 文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码：

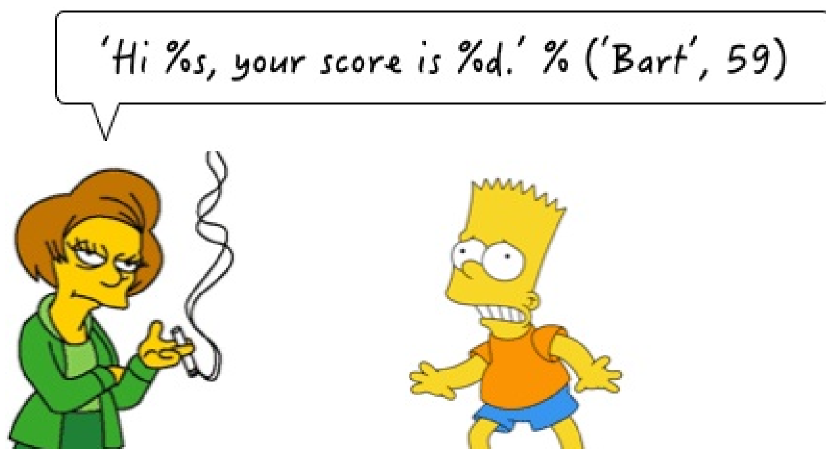


如果 `.py` 文件本身使用UTF-8编码，并且也申明了 `# -*- coding: utf-8 -*-` ，打开命令提示符测试就可以正常显示中文：



格式化

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似 `'亲爱的xxx你好！你xx月的话费是xx，余额是xx'` 之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。



在Python中，采用的格式化方式和C语言是一致的，用 `%` 实现，举例如下：

```
>>> 'Hello, %s' % 'world'
'Hello, world'
>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
'Hi, Michael, you have $1000000.'
```

你可能猜到了， `%` 运算符就是用来格式化字符串的。在字符串内部， `%s` 表示用字符串替换， `%d` 表示用整数替换，有几个 `%?` 占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个 `%?` ，括号可以省略。

常见的占位符有：

| %d | 整数 | | %f | 浮点数 | | %s | 字符串 | | %x | 十六进制整数 |

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

```
>>> '%2d-%02d' % (3, 1)
' 3-01'
>>> '%.2f' % 3.1415926
'3.14'
```

如果你不太确定应该用什么， `%s` 永远起作用，它会把任何数据类型转换为字符串：

```
>>> 'Age: %s. Gender: %s' % (25, True)
'Age: 25\. Gender: True'
```

有些时候，字符串里面的 `%` 是一个普通字符怎么办？这个时候就需要转义，用 `%%` 来表示一个 `%`：

```
>>> 'growth rate: %d %%' % 7
'growth rate: 7 %'
```

练习

小明的成绩从去年的72分提升到了今年的85分，请计算小明成绩提升的百分点，并用字符串格式化显示出 `'xx.x%'`，只保留小数点后1位：

```
# -*- coding: utf-8 -*-

s1 = 72
s2 = 85

r = ???
print('???' % r)
```

小结

Python 3的字符串使用Unicode，直接支持多语言。

str和bytes互相转换时，需要指定编码。最常用的编码是UTF-8。Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

```
>>> '中文'.encode('gb2312')
'\xd6\xd0\xce\xc4'
```

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用UTF-8编码。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

参考源码

the_string.py

使用list和tuple

list

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

```
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
```

变量 `classmates` 就是一个list。用 `len()` 函数可以获得list元素的个数：

```
>>> len(classmates)
3
```

用索引来访问list中每一个位置的元素，记得索引是从 `0` 开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是 `len(classmates) - 1` 。

如果要取最后一个元素，除了计算索引位置外，还可以用 `-1` 做索引，直接获取最后一个元素：

```
>>> classmates[-1]
'Tracy'
```

以此类推，可以获取倒数第2个、倒数第3个：

```
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为 `1` 的位置：

```
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除list末尾的元素，用 `pop()` 方法：

```
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

要删除指定位置的元素，用 `pop(i)` 方法，其中 `i` 是索引位置：

```
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

list里面的元素的数据类型也可以不同，比如：

```
>>> L = ['Apple', 123, True]
```

list元素也可以是另一个list，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
>>> len(s)
4
```

要注意 `s` 只有4个元素，其中 `s[2]` 又是一个list，如果拆开写就更容易理解了：

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
```

要拿到 `'php'` 可以写 `p[1]` 或者 `s[2][1]` ，因此 `s` 可以看成是一个二维数组，类似的还有三维、四维……数组，不过很少用到。

如果一个list中一个元素也没有，就是一个空的list，它的长度为0：

```
>>> L = []
>>> len(L)
0
```

tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用 `classmates[0]` ， `classmates[-1]` ，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

tuple的陷阱：当你定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来，比如：

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的tuple，可以写成 `()` ：

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有1个元素的tuple，如果你这么定义：

```
>>> t = (1)
>>> t
1
```

定义的不是tuple，是 `1` 这个数！这是因为括号 `()` 既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是 `1` 。

所以，只有1个元素的tuple定义时必须加一个逗号 `,` ，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

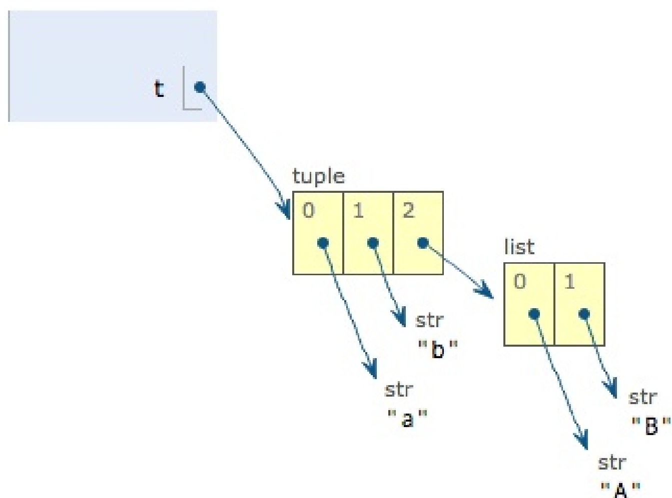
Python在显示只有1个元素的tuple时，也会加一个逗号 `,` ，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

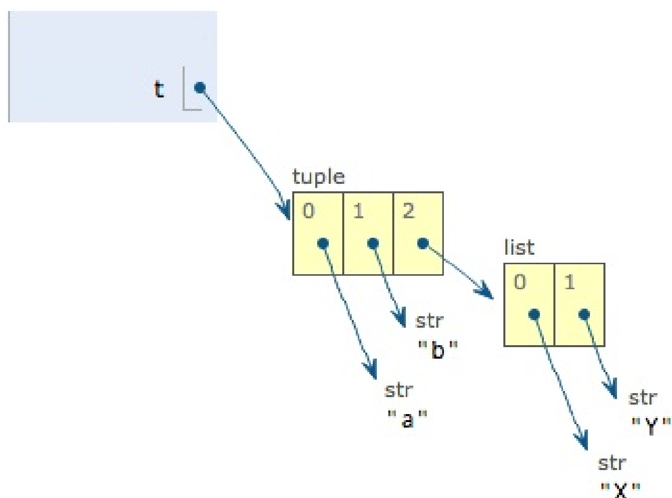
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

这个tuple定义的时候有3个元素，分别是 `'a'` ， `'b'` 和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素 `'A'` 和 `'B'` 修改为 `'X'` 和 `'Y'` 后，tuple变为：



表面上看，tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向 `'a'` ，就不能改成指向 `'b'` ，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

练习

请用索引取出下面list的指定元素：

```
# -*- coding: utf-8 -*-

L = [
    ['Apple', 'Google', 'Microsoft'],
    ['Java', 'Python', 'Ruby', 'PHP'],
    ['Adam', 'Bart', 'Lisa']
]

# 打印Apple:
print(?)
# 打印Python:
print(?)
# 打印Lisa:
print(?)
```

小结

list和tuple是Python内置的有序集合，一个可变，一个不可变。根据需要来选择使用它们。

参考源码

the_list.py

the_tuple.py

条件判断

条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用 `if` 语句实现：

```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

根据Python的缩进规则，如果 `if` 语句判断是 `True` ，就把缩进的两行print语句执行了，否则，什么也不做。

也可以给 `if` 添加一个 `else` 语句，意思是，如果 `if` 判断是 `False` ，不要执行 `if` 的内容，去把 `else` 执行了：

```
age = 3
if age >= 18:
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
```

注意不要少写了冒号 `:` 。

当然上面的判断是很粗略的，完全可以用 `elif` 做更细致的判断：

```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

`elif` 是 `else if` 的缩写，完全可以有多个 `elif` ，所以 `if` 语句的完整形式就是：

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

`if` 语句执行有个特点，它是从上往下判断，如果在某个判断上是 `True` ，把该判断对应的语句执行后，就忽略掉剩下的 `elif` 和 `else` ，所以，请测试并解释为什么下面的程序打印的是 `teenager` ：

```
age = 20
if age >= 6:
    print('teenager')
elif age >= 18:
    print('adult')
else:
    print('kid')
```

`if` 判断条件还可以简写，比如写：

```
if x:
    print('True')
```

只要 `x` 是非零数值、非空字符串、非空list等，就判断为 `True` ，否则为 `False` 。

再议 input

最后看一个有问题的条件判断。很多同学会用 `input()` 读取用户的输入，这样可以自己输入，程序运行得更有意思：

```
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

输入 `1982` ，结果报错：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

这是因为 `input()` 返回的数据类型是 `str` ， `str` 不能直接和整数比较，必须先把 `str` 转换成整数。Python提供了 `int()` 函数来完成这件事情：

```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

再次运行，就可以得到正确地结果。但是，如果输入 `abc` 呢？又会得到一个错误信息：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

原来`int()`函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

练习

小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：

- 低于18.5：过轻
- 18.5-25：正常
- 25-28：过重
- 28-32：肥胖
- 高于32：严重肥胖

用`if-elif`判断并打印结果：

```
# -*- coding: utf-8 -*-

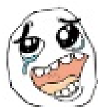
height = 1.75
weight = 80.5

bmi = ???
if ???:
    pass
```

小结

条件判断可以让计算机自己做选择，Python的if...elif...else很灵活。

if salary >= 10000:

print 

elif salary >=5000:

print 

else:

print 

参考源码

do_if.py

循环

循环

要计算1+2+3，我们可以直接写表达式：

```
>>> 1 + 2 + 3
6
```

要计算1+2+3+...+10，勉强也能写出来。

但是，要计算1+2+3+...+10000，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)
```

执行这段代码，会依次打印 `names` 的每一个元素：

```
Michael
Bob
Tracy
```

所以 `for x in ...` 循环就是把每个元素代入变量 `x` ，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个 `sum` 变量做累加：

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个 `range()` 函数，可以生成一个整数序列，再通过 `list()` 函数可以转换为list。比如 `range(5)` 生成的序列是从0开始小于5的整数：

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

`range(101)` 就可以生成0-100的整数序列，计算如下：

```
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
```

在循环内部变量 `n` 不断自减，直到变为 `-1` 时，不再满足while条件，循环退出。

练习

请利用循环依次对list中的每个名字打印出 `Hello, xxx!` ：

```
# -*- coding: utf-8 -*-
L = ['Bart', 'Lisa', 'Adam']
```

小结

循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用 `Ctrl+C` 退出程序，或者强制结束Python进程。

请试写一个死循环程序。

参考源码

do_for.py

do_while.py

使用dict和set

dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如 `'Michael'` ，dict在内部就可以直接计算出 `Michael` 对应的存放成绩的“页码”，也就是 `95` 这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果key不存在，dict就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

要避免key不存在的错误，有两种办法，一是通过 `in` 判断key是否存在：

```
>>> 'Thomas' in d
False
```

二是通过dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value：

```
>>> d.get('Thomas')
>>> d.get('Thomas', -1)
-1
```

注意：返回 `None` 的时候Python的交互式命令行不显示结果。

要删除一个key，用 `pop(key)` 方法，对应的value也会从dict中删除：

```
>>> d.pop('Bob')
75
>>> d
{'Michael': 95, 'Tracy': 85}
```

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而增加；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是不可变对象。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

```
>>> key = [1, 2, 3]
>>> d[key] = 'a list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

```
>>> s = set([1, 2, 3])
>>> s
{1, 2, 3}
```

注意，传入的参数 `[1, 2, 3]` 是一个list，而显示的 `{1, 2, 3}` 只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。。

重复元素在set中自动被过滤：

```
>>> s = set([1, 1, 2, 2, 3, 3])
>>> s
{1, 2, 3}
```

通过 `add(key)` 方法可以添加元素到set中，可以重复添加，但不会有效果：

```
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

通过 `remove(key)` 方法可以删除元素：

```
>>> s.remove(4)
>>> s
{1, 2, 3}
```

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
{2, 3}
>>> s1 | s2
{1, 2, 3, 4}
```

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。试试把list放入set，看看是否会报错。

再议不可变对象

上面我们讲了，str是不变对象，而list是可变对象。

对于可变对象，比如list，对list进行操作，list内部的内容是会变化的，比如：

```
>>> a = ['c', 'b', 'a']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

而对于不可变对象，比如str，对str进行操作呢：

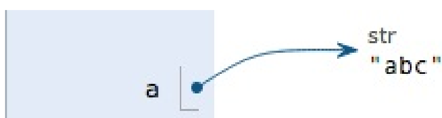
```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

虽然字符串有个 `replace()` 方法，也确实变出了 `'Abc'`，但变量 `a` 最后仍是 `'abc'`，应该怎么理解呢？

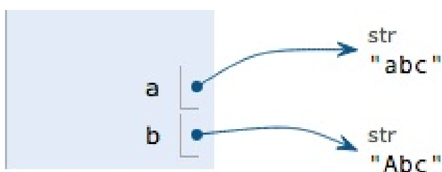
我们先把代码改成下面这样：

```
>>> a = 'abc'
>>> b = a.replace('a', 'A')
>>> b
'Abc'
>>> a
'abc'
```

要始终牢记的是，`a` 是变量，而 `'abc'` 才是字符串对象！有些时候，我们经常说，对象 `a` 的内容是 `'abc'`，但其实是指，`a` 本身是一个变量，它指向的对象的内容才是 `'abc'`：



当我们调用 `a.replace('a', 'A')` 时，实际上调用方法 `replace` 是作用在字符串对象 `'abc'` 上的，而这个方法虽然名字叫 `replace`，但却没有改变字符串 `'abc'` 的内容。相反，`replace` 方法创建了一个新字符串 `'Abc'` 并返回，如果我们用变量 `b` 指向该新字符串，就容易理解了，变量 `a` 仍指向原有的字符串 `'abc'`，但变量 `b` 却指向新字符串 `'Abc'` 了：



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把 `(1, 2, 3)` 和 `(1, [2, 3])` 放入dict或set中，并解释结果。

参考源码

the_dict.py

the_set.py

函数

我们知道圆的面积计算公式为：

S = πr²

当我们知道半径 `r` 的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把 `3.14` 改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x`，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：`1 + 2 + 3 + ... + 100`，写起来十分不方便，于是数学家发明了求和符号∑，可以把 `1 + 2 + 3 + ... + 100` 记作：

100

∑n

n=1

这种抽象记法非常强大，因为我们看到 ∑ 就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

∑(n²+1)

n=1

还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数 `abs` ，只有一个参数。可以直接从Python的官方网站查看文档：

http://docs.python.org/3/library/functions.html#abs

也可以在交互式命令行通过 `help(abs)` 查看 `abs` 函数的帮助信息。

调用 `abs` 函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

调用函数的时候，如果传入的参数数量不对，会报 `TypeError` 的错误，并且Python会明确地告诉你： `abs()` 有且仅有1个参数，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 `TypeError` 的错误，并且给出错误信息： `str` 是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而 `max` 函数 `max()` 可以接收任意多个参数，并返回最大的那个：

```
>>> max(1, 2)
2
>>> max(2, 3, 1, -5)
3
```

数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如 `int()` 函数可以把其他数据类型转换为整数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> str(100)
'100'
>>> bool(1)
True
>>> bool('')
False
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量a指向abs函数
>>> a(-1) # 所以也可以通过a调用abs函数
1
```

练习

请利用Python内置的`hex()`函数把一个整数转换成十六进制表示的字符串：

```
# -*- coding: utf-8 -*-

n1 = 255
n2 = 1000

print(???)
```

小结

调用Python的函数，需要根据函数定义，传入正确的参数。如果函数调用出错，一定要学会看错误信息，所以英文很重要！

参考源码

call_func.py

定义函数

在Python中，定义一个函数要使用 `def` 语句，依次写出函数名、括号、括号中的参数和冒号 `:` ，然后，在缩进块中编写函数体，函数的返回值用 `return` 语句返回。

我们以自定义一个求绝对值的 `my_abs` 函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

请自行测试并调用 `my_abs` 看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后也会返回结果，只是结果为 `None` 。

`return None` 可以简写为 `return` 。

在Python交互环境中定义函数时，注意Python会出现 `...` 的提示。函数定义结束后需要按两次回车重新回到 `>>>` 提示符下：

http://michaelliao.gitcafe.io/video/py/def-myabs.mp4

如果你已经把 `my_abs()` 的函数定义保存为 `abstest.py` 文件了，那么，可以在该文件的当前目录下启动Python解释器，用 `from abstest import my_abs` 来导入 `my_abs()` 函数，注意 `abstest` 是文件名（不含 `.py` 扩展名）：

http://michaelliao.gitcafe.io/video/py/import-abstest.mp4

`import` 的用法在后续模块一节中会详细介绍。

空函数

如果想定义一个什么事也不做的空函数，可以用 `pass` 语句：

```
def nop():
    pass
```

`pass` 语句什么都不做，那有什么用？实际上 `pass` 可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个 `pass` ，让代码能运行起来。

`pass` 还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了 `pass` ，代码运行就会有语法错误。

参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出 `TypeError` ：

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes 1 positional argument but 2 were given
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试 `my_abs` 和内置函数 `abs` 的差别：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in my_abs
TypeError: unorderable types: str() >= int()
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时，内置函数 `abs` 会检查出参数错误，而我们定义的 `my_abs` 没有参数检查，会导致 `if` 语句出错，出错信息和 `abs` 不一样。所以，这个函数定义不够完善。

让我们修改一下 `my_abs` 的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数 `isinstance()` 实现：

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续讲到。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

```
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

`import math` 语句表示导入 `math` 包，并允许后续代码引用 `math` 包里的 `sin` 、 `cos` 等函数。

然后，我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print(x, y)
151.96152422706632 70.0
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print(r)
(151.96152422706632, 70.0)
```

原来返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用 `return` 随时返回函数结果；

函数执行完毕也没有`return`语句时，自动`return None`。

函数可以同时返回多个值，但其实就是一个tuple。

练习

请定义一个函数`quadratic(a, b, c)`，接收3个参数，返回一元二次方程：

ax² + bx + c = 0

的两个解。

提示：计算平方根可以调用`math.sqrt()`函数：

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

```
# -*- coding: utf-8 -*-

import math

def quadratic(a, b, c):

    pass

# 测试:
print(quadratic(2, 3, 1)) # => (-0.5, -1.0)
print(quadratic(1, 3, -4)) # => (1.0, -4.0)
```

参考源码

def_func.py

函数的参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

位置参数

我们先写一个计算x²的函数：

```
def power(x):
    return x * x
```

对于 `power(x)` 函数，参数 `x` 就是一个位置参数。

当我们调用 `power` 函数时，必须传入有且仅有的一个参数 `x` ：

```
>>> power(5)
25
>>> power(15)
225
```

现在，如果我们要计算x³怎么办？可以再定义一个 `power3` 函数，但是如果要计算x⁴、x⁵……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 `power(x)` 修改为 `power(x, n)` ，用来计算xⁿ，说干就干：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

对于这个修改后的 `power(x, n)` 函数，可以计算任意n次方：

```
>>> power(5, 2)
25
>>> power(5, 3)
125
```

修改后的 `power(x, n)` 函数有两个参数： `x` 和 `n` ，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 `x` 和 `n` 。

默认参数

新的 `power(x, n)` 函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python的错误信息很明确：调用函数 `power()` 缺少了一个位置参数 `n` 。

这个时候，默认参数就排上用场了。由于我们经常计算x²，所以，完全可以把第二个参数n的默认值设定为2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用`power(5)`时，相当于调用`power(5, 2)`：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于`n > 2`的其他情况，就必须明确地传入n，比如`power(5, 3)`。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入`name`和`gender`两个参数：

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用`enroll()`函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)` ，意思是，除了 `name` ， `gender` 这两个参数外，最后1个参数应用在参数 `age` 上， `city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')` ，意思是， `city` 参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个 `END` 再返回：

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 `add_end()` 时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是 `[]` ，但是函数似乎每次都“记住了”上次添加了 `'END'` 后的list。

原因解释如下：

Python函数在定义的时候，默认参数 `L` 的值就被计算出来了，即 `[]` ，因为默认参数 `L` 也是一个变量，它指向对象 `[]` ，每次调用该函数，如果改变了 `L` 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的 `[]` 了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 `None` 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计 `str` 、 `None` 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字a，b，c……，请计算a² + b² + c² + ……。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个list或tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个 `*` 号。在函数内部，参数 `numbers` 接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个 `*` 号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

`*nums` 表示把 `nums` 这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

函数 `person` 除了必选参数 `name` 和 `age` 外，还接受关键字参数 `kw` 。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在 `person` 函数里，我们保证能接收到 `name` 和 `age` 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=extra['city'], job=extra['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **extra)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

`**extra` 表示把 `extra` 这个dict的所有key-value用关键字参数传入到函数的 `**kw` 参数， `kw` 将获得一个dict，注意 `kw` 获得的dict是 `extra` 的一份拷贝，对 `kw` 的改动不会影响到函数外的 `extra` 。

命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 `kw` 检查。

仍以 `person()` 函数为例，我们希望检查是否有 `city` 和 `job` 参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 `city` 和 `job` 作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数 `**kw` 不同，命名关键字参数需要一个特殊分隔符 `*` ， `*` 后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数名 `city` 和 `job` ，Python解释器把这4个参数均视为位置参数，但 `person()` 函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 `city` 具有默认值，调用时，可不传入 `city` 参数：

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意， `*` 不是参数，而是特殊分隔符。如果缺少 `*` ，Python解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *，city和job被视为位置参数
    pass
```

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用，除了可变参数无法和命名关键字参数混合。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数/命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似 `func(*args, **kw)` 的形式调用它，无论它的参数是如何定义的。

小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

`*args` 是可变参数，args接收的是一个tuple；

`**kw` 是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：`func(1, 2, 3)`，又可以先组装list或tuple，再通过 `*args` 传入：`func(*(1, 2, 3))`；

关键字参数既可以直接传入：`func(a=1, b=2)`，又可以先组装dict，再通过 `**kw` 传入：`func(**{'a': 1, 'b': 2})`。

使用 `*args` 和 `**kw` 是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数不要忘了写分隔符 `*`，否则定义的将是位置参数。

参考源码

var_args.py

kw_args.py

递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 `n! = 1 x 2 x 3 x ... x n`，用函数 `fact(n)` 表示，可以看出：

fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n

所以，`fact(n)` 可以表示为 `n x fact(n-1)`，只有n=1时需要特殊处理。

于是，`fact(n)` 用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217
```

如果我们计算 `fact(5)`，可以根据函数定义看到计算过程如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(1000)` ：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)` 仅返回递归函数本身，`num - 1` 和 `num * product` 在函数调用前就会被计算，不影响函数调用。

`fact(5)` 对应的 `fact_iter(5, 1)` 的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

练习

汉诺塔的移动可以用递归函数非常简单地实现。

请编写 `move(n, a, b, c)` 函数，它接收参数 `n`，表示3个柱子A、B、C中第1个柱子A的盘子数量，然后打印出把所有盘子从A借助B移动到C的方法，例如：

```
def move(n, a, b, c):

    pass

# 期待输出:
# A --> C
# A --> B
# C --> B
# A --> C
# B --> A
# B --> C
# A --> C
move(3, 'A', 'B', 'C')
```

参考源码

recur.py

高级特性

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个`1, 3, 5, 7, ..., 99`的列表，可以通过循环实现：

```
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2
```

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，1行代码能实现的功能，决不写5行代码。请始终牢记，代码越少，开发效率越高。

切片

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

取前3个元素，应该怎么做？

笨办法：

```
>>> [L[0], L[1], L[2]]
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

```
>>> r = []
>>> n = 3
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

`L[0:3]` 表示，从索引 `0` 开始取，直到索引 `3` 为止，但不包括索引 `3` 。即索引 `0` ， `1` ， `2` ，正好是3个元素。

如果第一个索引是 `0` ，还可以省略：

```
>>> L[:3]
['Michael', 'Sarah', 'Tracy']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然Python支持 `L[-1]` 取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

记住倒数第一个元素的索引是 `-1` 。

切片操作十分有用。我们先创建一个0-99的数列：

```
>>> L = list(range(100))
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前10个数：

```
>>> L[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

后10个数：

```
>>> L[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

前11-20个数：

```
>>> L[10:20]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前10个数，每两个取一个：

```
>>> L[:10:2]
[0, 2, 4, 6, 8]
```

所有数，每5个取一个：

```
>>> L[::5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80,
```

甚至什么都不写，只写 `[:]` 就可以原样复制一个list：

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

```
>>> (0, 1, 2, 3, 4, 5)[:3]
(0, 1, 2)
```

字符串 `'xxx'` 也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

```
>>> 'ABCDEFG'[:3]
'ABC'
>>> 'ABCDEFG'[::2]
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

小结

有了切片操作，很多地方循环就不再需要了。Python的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

参考源码

do_slice.py

迭代

如果给定一个list或tuple，我们可以通过 `for` 循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过 `for ... in` 来完成的，而很多语言比如C或者Java，迭代list是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {
    n = list[i];
}
```

可以看出，Python的 `for` 循环抽象程度要高于Java的 `for` 循环，因为Python的 `for` 循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print(key)
...
a
c
b
```

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用 `for value in d.values()` ，如果要同时迭代key和value，可以用 `for k, v in d.items()` 。

由于字符串也是可迭代对象，因此，也可以作用于 `for` 循环：

```
>>> for ch in 'ABC':
...     print(ch)
...
A
B
C
```

所以，当我们使用 `for` 循环时，只要作用于一个可迭代对象， `for` 循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str是否可迭代
True
>>> isinstance([1,2,3], Iterable) # list是否可迭代
True
>>> isinstance(123, Iterable) # 整数是否可迭代
False
```

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的 `enumerate` 函数可以把一个list变成索引-元素对，这样就可以在 `for` 循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):
...     print(i, value)
...
0 A
1 B
2 C
```

上面的 `for` 循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:
...     print(x, y)
...
1 1
2 4
3 9
```

小结

任何可迭代对象都可以作用于 `for` 循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用 `for` 循环。

参考源码

do_iter.py

列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` 可以用 `list(range(1, 11))` ：

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成 `[1x1, 2x2, 3x3, ..., 10x10]` 怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...    L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素 `x * x` 放到前面，后面跟 `for` 循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop',
```

`for` 循环其实可以同时使用两个甚至多个变量，比如 `dict` 的 `items()` 可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.items():
...     print(k, '=', v)
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.items()]
['y=B', 'x=A', 'z=C']
```

最后把一个list中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

练习

如果list中既包含字符串，又包含整数，由于非字符串类型没有 `lower()` 方法，所以列表生成式会报错：

```
>>> L = ['Hello', 'World', 18, 'Apple', None]
>>> [s.lower() for s in L]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <listcomp>
AttributeError: 'int' object has no attribute 'lower'
```

使用内建的 `isinstance` 函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
False
```

请修改列表生成式，通过添加 `if` 语句保证列表生成式能正确地执行：

```
# -*- coding: utf-8 -*-

L1 = ['Hello', 'World', 18, 'Apple', None]

L2 = ???

# 期待输出: ['hello', 'world', 'apple']
print(L2)
```

小结

运用列表生成式，可以快速生成list，可以通过一个list推导出另一个list，而代码却十分简洁。

参考源码

do_listcompr.py

生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的 `[]` 改成 `()` ，就创建了一个generator：

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建 `L` 和 `g` 的区别仅在于最外层的 `[]` 和 `()` ， `L` 是一个list，而 `g` 是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过 `next()` 函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

我们讲过，generator保存的是算法，每次调用 `next(g)` ，就计算出 `g` 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 `StopIteration` 的错误。

当然，上面这种不断调用 `next(g)` 实在是太变态了，正确的方法是使用 `for` 循环，因为generator也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...
0
1
4
9
16
25
36
49
64
81
```

所以，我们创建了一个generator后，基本上永远不会调用 `next()` ，而是通过 `for` 循环来迭代它，并且不需要关心 `StopIteration` 的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

上面的函数可以输出斐波那契数列的前N个数：

```
>>> fib(6)
1
1
2
3
5
8
'done'
```

仔细观察，可以看出，`fib`函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把`fib`函数变成generator，只需要把`print(b)`改为`yield b`就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

这就是定义generator的另一种方法。如果一个函数定义中包含`yield`关键字，那么这个函数就不再是一个普通函数，而是一个generator：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到`return`语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用`next()`的时候执行，遇到`yield`语句返回，再次执行时从上次返回的`yield`语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
    print('step 3')
    yield(5)
```

调用该generator时，首先要生成一个generator对象，然后用 `next()` 函数不断获得下一个返回值：

```
>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到， `odd` 不是普通函数，而是generator，在执行过程中，遇到 `yield` 就中断，下次又继续执行。执行3次 `yield` 后，已经没有 `yield` 可以执行了，所以，第4次调用 `next(o)` 就报错。

回到 `fib` 的例子，我们在循环过程中不断调用 `yield` ，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用 `next()` 来获取下一个返回值，而是直接使用 `for` 循环来迭代：

```
>>> for n in fib(6):
...     print(n)
...
1
1
2
3
5
8
```

但是用 `for` 循环调用generator时，发现拿不到generator的 `return` 语句的返回值。如果想要拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的 `value` 中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

关于如何捕获错误，后面的错误处理还会详细讲解。

练习

杨辉三角定义如下：

```
 1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

把每一行看做一个list，试写一个generator，不断输出下一行的list：

```
# -*- coding: utf-8 -*-

def triangles():

    pass

# 期待输出:
# [1]
# [1, 1]
# [1, 2, 1]
# [1, 3, 3, 1]
# [1, 4, 6, 4, 1]
# [1, 5, 10, 10, 5, 1]
# [1, 6, 15, 20, 15, 6, 1]
# [1, 7, 21, 35, 35, 21, 7, 1]
# [1, 8, 28, 56, 70, 56, 28, 8, 1]
# [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
n = 0
for t in triangles():
    print(t)
    n = n + 1
    if n == 10:
        break
```

小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在 `for` 循环的过程中不断计算出下一个元素，并在适当的条件结束 `for` 循环。对于函数改成的generator来说，遇到 `return` 语句或者执行到函数体最后一行语句，就是结束generator的指令， `for` 循环随之结束。

请注意区分普通函数和generator函数，普通函数调用直接返回结果：

```
>>> r = abs(6)
>>> r
6
```

generator函数的“调用”实际返回一个generator对象：

```
>>> g = fib(6)
>>> g
<generator object fib at 0x1022ef948>
```

参考源码

do_generator.py

迭代器

我们已经知道，可以直接作用于 `for` 循环的数据类型有以下几种：

一类是集合数据类型，如 `list` 、 `tuple` 、 `dict` 、 `set` 、 `str` 等；

一类是 `generator` ，包括生成器和带 `yield` 的generator function。

这些可以直接作用于 `for` 循环的对象统称为可迭代对象： `Iterable` 。

可以使用 `isinstance()` 判断一个对象是否是 `Iterable` 对象：

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True
>>> isinstance({}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
>>> isinstance(100, Iterable)
False
```

而生成器不但可以作用于 `for` 循环，还可以被 `next()` 函数不断调用并返回下一个值，直到最后抛出 `StopIteration` 错误表示无法继续返回下一个值了。

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器： `Iterator` 。

可以使用 `isinstance()` 判断一个对象是否是 `Iterator` 对象：

```
>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

生成器都是 `Iterator` 对象，但 `list` 、 `dict` 、 `str` 虽然是 `Iterable` ，却不是 `Iterator` 。

把 `list` 、 `dict` 、 `str` 等 `Iterable` 变成 `Iterator` 可以使用 `iter()` 函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

你可能会问，为什么 `list` 、 `dict` 、 `str` 等数据类型不是 `Iterator` ？

这是因为Python的 `Iterator` 对象表示的是一个数据流，Iterator对象可以被 `next()` 函数调用并不断返回下一个数据，直到没有数据时抛出 `StopIteration` 错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 `next()` 函数实现按需计算下一个数据，所以 `Iterator` 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

`Iterator` 甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

小结

凡是可作用于 `for` 循环的对象都是 `Iterable` 类型；

凡是可作用于 `next()` 函数的对象都是 `Iterator` 类型，它们表示一个惰性计算的序列；

集合数据类型如 `list` 、 `dict` 、 `str` 等是 `Iterable` 但不是 `Iterator` ，不过可以通过 `iter()` 函数获得一个 `Iterator` 对象。

Python的 `for` 循环本质上就是通过不断调用 `next()` 函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

实际上完全等价于：

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

参考源码

do_iter.py

函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

高阶函数

高阶函数英文叫Higher-order function。什么是高阶函数？我们以实际代码为例子，一步一步深入概念。

变量可以指向函数

以Python内置的求绝对值的函数 `abs()` 为例，调用该函数用以下代码：

```
>>> abs(-10)
10
```

但是，如果只写 `abs` 呢？

```
>>> abs
<built-in function abs>
```

可见， `abs(-10)` 是函数调用，而 `abs` 是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量 `f` 现在已经指向了 `abs` 函数本身。直接调用 `abs()` 函数和调用变量 `f()` 完全相同。

函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于 `abs()` 这个函数，完全可以把函数名 `abs` 看成变量，它指向一个可以计算绝对值的函数！

如果把 `abs` 指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 `abs` 指向 `10` 后，就无法通过 `abs(-10)` 调用该函数了！因为 `abs` 这个变量已经不指向求绝对值函数而是指向一个整数 `10` ！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复 `abs` 函数，请重启Python交互环境。

注：由于 `abs` 函数实际上是定义在 `__builtin__` 模块中的，所以要让修改 `abs` 变量的指向在其它模块也生效，要用 `__builtin__.abs = 10` 。

传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

当我们调用`add(-5, 6, abs)`时，参数`x`，`y`和`f`分别接收`-5`，`6`和`abs`，根据函数定义，我们可以推导计算过程为：

```
x = -5
y = 6
f = abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
return 11
```

用代码验证一下：

```
>>> add(-5, 6, abs)
11
```

编写高阶函数，就是让函数的参数能够接收别的函数。

小结

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

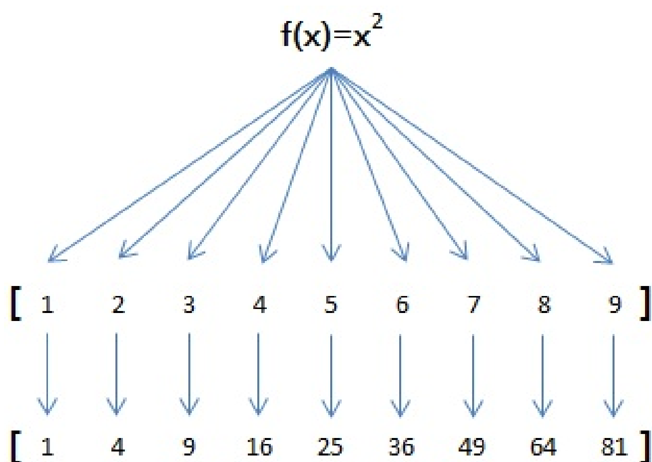
map/reduce

Python内建了 `map()` 和 `reduce()` 函数。

如果你读过Google的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。

我们先看map。 `map()` 函数接收两个参数，一个是函数，一个是 `Iterable` ， `map` 将传入的函数依次作用到序列的每个元素，并把结果作为新的 `Iterator` 返回。

举例说明，比如我们有一个函数f(x)=x²，要把这个函数作用在一个list `[1, 2, 3, 4, 5, 6, 7, 8, 9]` 上，就可以用 `map()` 实现如下：



现在，我们用Python代码实现：

```
>>> def f(x):
...     return x * x
...
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`map()` 传入的第一个参数是 `f` ，即函数对象本身。由于结果 `r` 是一个 `Iterator` ， `Iterator` 是惰性序列，因此通过 `list()` 函数让它把整个序列都计算出来并返回一个list。

你可能会想，不需要 `map()` 函数，写一个循环，也可以计算出结果：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
```

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，`map()` 作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x²，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

```
>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

只需要一行代码。

再看 `reduce` 的用法。 `reduce` 把一个函数作用在一个序列 `[x1, x2, x3, ...]` 上，这个函数必须接收两个参数， `reduce` 把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用 `reduce` 实现：

```
>>> from functools import reduce
>>> def add(x, y):
...     return x + y
...
>>> reduce(add, [1, 3, 5, 7, 9])
25
```

当然求和运算可以直接用Python内建函数 `sum()` ，没必要动用 `reduce` 。

但是如果要把序列 `[1, 3, 5, 7, 9]` 变换成整数 `13579` ， `reduce` 就可以派上用场：

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

这个例子本身没多大用处，但是，如果考虑到字符串 `str` 也是一个序列，对上面的例子稍加改动，配合 `map()` ，我们就可以写出把 `str` 转换为 `int` 的函数：

```
>>> from functools import reduce
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
...     return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6
...
>>> reduce(fn, map(char2num, '13579'))
13579
```

整理成一个 `str2int` 的函数就是：

```
from functools import reduce

def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6
    return reduce(fn, map(char2num, s))
```

还可以用lambda函数进一步简化成：

```
from functools import reduce

def char2num(s):
    return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6,

def str2int(s):
    return reduce(lambda x, y: x * 10 + y, map(char2num, s))
```

也就是说，假设Python没有提供 `int()` 函数，你完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

lambda函数的用法在后面介绍。

练习

利用 `map()` 函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入： `['adam', 'LISA', 'barT']` ，输出： `['Adam', 'Lisa', 'Bart']` ：

```
# -*- coding: utf-8 -*-

def normalize(name):
    pass

# 测试:
L1 = ['adam', 'LISA', 'barT']
L2 = list(map(normalize, L1))
print(L2)
```

Python提供的 `sum()` 函数可以接受一个list并求和，请编写一个 `prod()` 函数，可以接受一个list并利用 `reduce()` 求积：

```
# -*- coding: utf-8 -*-

from functools import reduce

def prod(L):

    pass

print('3 * 5 * 7 * 9 =', prod([3, 5, 7, 9]))
```

利用 `map` 和 `reduce` 编写一个 `str2float` 函数，把字符串 `'123.456'` 转换成浮点数 `123.456` ：

```
# -*- coding: utf-8 -*-

from functools import reduce

def str2float(s):

    pass

print('str2float(\'123.456\') =', str2float('123.456'))
```

参考代码

do_map.py

do_reduce.py

filter

Python内建的 `filter()` 函数用于过滤序列。

和 `map()` 类似， `filter()` 也接收一个函数和一个序列。和 `map()` 不同的时， `filter()` 把传入的函数依次作用于每个元素，然后根据返回值是 `True` 还是 `False` 决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
# 结果: [1, 5, 9, 15]
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):
    return s and s.strip()

list(filter(not_empty, ['A', '', 'B', None, 'C', '  ']))
# 结果: ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到 `filter()` 函数返回的是一个 `Iterator` ，也就是一个惰性序列，所以要强迫 `filter()` 完成计算结果，需要用 `list()` 函数获得所有结果并返回list。

用filter求素数

计算素数的一个方法是埃氏筛法，它的算法理解起来非常简单：

首先，列出从 `2` 开始的所有自然数，构造一个序列：

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...

取序列的第一个数 `2` ，它一定是素数，然后用 `2` 把序列的 `2` 的倍数筛掉：

3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~, ...

取新序列的第一个数 `3` ，它一定是素数，然后用 `3` 把序列的 `3` 的倍数筛掉：

5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~, ...

取新序列的第一个数 `5` ，然后用 `5` 把序列的 `5` 的倍数筛掉：

7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~, ...

不断筛下去，就可以得到所有的素数。

用Python来实现这个算法，可以先构造一个从 `3` 开始的奇数序列：

```
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n
```

注意这是一个生成器，并且是一个无限序列。

然后定义一个筛选函数：

```
def _not_divisible(n):
    return lambda x: x % n > 0
```

最后，定义一个生成器，不断返回下一个素数：

```
def primes():
    yield 2
    it = _odd_iter() # 初始序列
    while True:
        n = next(it) # 返回序列的第一个数
        yield n
        it = filter(_not_divisible(n), it) # 构造新序列
```

这个生成器先返回第一个素数 `2` ，然后，利用 `filter()` 不断产生筛选后的新的序列。

由于 `primes()` 也是一个无限序列，所以调用时需要设置一个退出循环的条件：

```
# 打印1000以内的素数:
for n in primes():
    if n < 1000:
        print(n)
    else:
        break
```

注意到 `Iterator` 是惰性计算的序列，所以我们可以用Python表示“全体自然数”，“全体素数”这样的序列，而代码非常简洁。

练习

回数是指从左向右读和从右向左读都是一样的数，例如 `12321` ， `909` 。请利用 `filter()` 滤掉非回数：

```
# -*- coding: utf-8 -*-

def is_palindrome(n):

    pass

# 测试:
output = filter(is_palindrome, range(1, 1000))
print(list(output))
```

小结

`filter()` 的作用是从一个序列中筛出符合条件的元素。由于 `filter()` 使用了惰性计算，所以只有在取 `filter()` 结果的时候，才会真正筛选并每次返回下一个筛出的元素。

参考源码

do_filter.py

prime_numbers.py

sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的 `sorted()` 函数就可以对list进行排序：

```
>>> sorted([36, 5, -12, 9, -21])
[-21, -12, 5, 9, 36]
```

此外， `sorted()` 函数也是一个高阶函数，它还可以接收一个 `key` 函数来实现自定义的排序，例如按绝对值大小排序：

```
>>> sorted([36, 5, -12, 9, -21], key=abs)
[5, 9, -12, -21, 36]
```

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。对比原始的list和经过 `key=abs` 处理过的list：

```
list = [36, 5, -12, 9, -21]

keys = [36, 5,  12, 9,  21]
```

然后 `sorted()` 函数按照keys进行排序，并按照对应关系返回list相应的元素：

```
keys排序结果 => [5, 9,  12,  21, 36]
                |  |    |    |   |
最终结果     => [5, 9, -12, -21, 36]
```

我们再看一个字符串排序的例子：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])
['Credit', 'Zoo', 'about', 'bob']
```

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于 `'Z' < 'a'` ，结果，大写字母 `Z` 会排在小写字母 `a` 的前面。

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个key函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给 `sorted` 传入key函数，即可实现忽略大小写的排序：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)
['about', 'bob', 'Credit', 'Zoo']
```

要进行反向排序，不必改动key函数，可以传入第三个参数 `reverse=True` ：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, revers
['Zoo', 'Credit', 'bob', 'about']
```

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

小结

`sorted()` 也是一个高阶函数。用 `sorted()` 排序的关键在于实现一个映射函数。

练习

假设我们用一组tuple表示学生名字和成绩：

```
L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]
```

请用 `sorted()` 对上述列表分别按名字排序：

```
# -*- coding: utf-8 -*-

L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]

def by_name(t):

    pass

L2 = sorted(L, key=by_name)
print(L2)
```

再按成绩从高到低排序：

```
# -*- coding: utf-8 -*-

L = [('Bob', 75), ('Adam', 92), ('Bart', 66), ('Lisa', 88)]


def by_score(t):
    pass

L2 = ???

print(L2)
```

参考源码

do_sorted.py

返回函数

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum` ，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `args` ，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用 `f1()` ， `f2()` 和 `f3()` 结果应该是 `1` ， `4` ， `9` ，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 `9` ！原因就在于返回的函数引用了变量 `i` ，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量 `i` 已经变成了 `3` ，因此最终结果为 `9` 。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
```

再看看结果：

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

缺点是代码较长，可利用lambda函数缩短代码。

小结

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

参考源码

return_func.py

匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以 `map()` 函数为例，计算 f(x)=x²时，除了定义一个 `f(x)` 的函数外，还可以直接传入匿名函数：

```
>>> list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 `lambda x: x * x` 实际上就是：

```
def f(x):
    return x * x
```

关键字 `lambda` 表示匿名函数，冒号前面的 `x` 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写 `return` ，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda: x * x + y * y
```

小结

Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print('2015-3-25')
...
>>> f = now
>>> f()
2015-3-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的 `log` ，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

```
@log
def now():
    print('2015-3-25')
```

调用 `now()` 函数，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志：

```
>>> now()
call now():
2015-3-25
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个decorator，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)`，因此，`wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个3层嵌套的decorator用法如下：

```
@log('execute')
def now():
    print('2015-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')` ，返回的是 `decorator` 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 `wrapper` 函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你去看经过decorator装饰之后的函数，它们的 `__name__` 已经从原来的 `'now'` 变成了 `'wrapper'` ：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 `wrapper()` 函数名字就是 `'wrapper'` ，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python内置的 `functools.wraps` 就是干这个事的，所以，一个完整的decorator的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的decorator：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出 `'begin call'` 和 `'end call'` 的日志。

再思考一下能否写出一个 `@log` 的decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

参考源码

decorator.py

偏函数

Python的 `functools` 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

`int()` 函数可以把字符串转换为整数，当仅传入字符串时， `int()` 函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 `int()` 函数还提供额外的 `base` 参数，默认值为 `10` 。如果传入 `base` 参数，就可以做N进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()` ，可以直接使用下面的代码创建一个新的函数 `int2` ：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 `2` ，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
1000000
```

最后，创建偏函数时，实际上可以接收函数对象、 `*args` 和 `**kw` 这3个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了int()函数的关键字参数 `base` ，也就是：

```
int2('10010')
```

相当于：

```
kw = { 'base': 2 }
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 `10` 作为 `*args` 的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)
max(*args)
```

结果为 `10` 。

小结

当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

参考源码

do_partial.py

模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

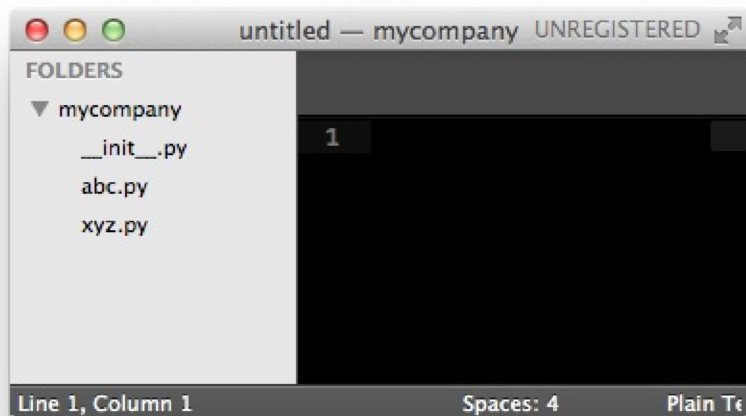
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点这里查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个 `abc.py` 的文件就是一个名字叫 `abc` 的模块，一个 `xyz.py` 的文件就是一个名字叫 `xyz` 的模块。

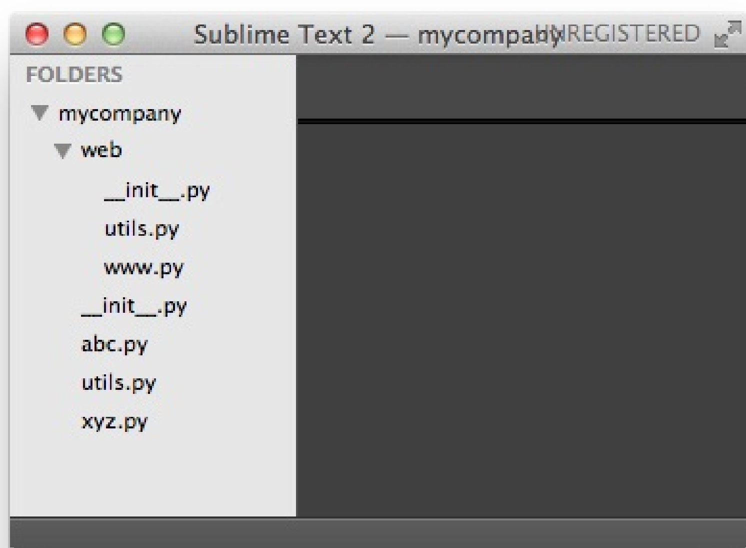
现在，假设我们的 `abc` 和 `xyz` 这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 `mycompany` ，按照如下目录存放：



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在， `abc.py` 模块的名字就变成了 `mycompany.abc` ，类似的， `xyz.py` 的模块名变成了 `mycompany.xyz` 。

请注意，每一个包目录下面都会有一个 `__init__.py` 的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。 `__init__.py` 可以是空文件，也可以有Python代码，因为 `__init__.py` 本身就是一个模块，而它的模块名就是 `mycompany` 。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



文件 `www.py` 的模块名就是 `mycompany.web.www` ，两个文件 `utils.py` 的模块名分别是 `mycompany.utils` 和 `mycompany.web.utils` 。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了 `sys` 模块，自己的模块就不可命名为 `sys.py` ，否则将无法导入系统自带的 `sys` 模块。

`mycompany.web` 也是一个模块，请指出该模块对应的.py文件。

使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的 `sys` 模块为例，编写一个 `hello` 的模块：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
            print('Hello, world!')
    elif len(args)==2:
        print('Hello, %s!' % args[1])
    else:
        print('Too many arguments!')

if __name__=='__main__':
    test()
```

第1行和第2行是标准注释，第1行注释可以让这个 `hello.py` 文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用 `__author__` 变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用 `sys` 模块的第一步，就是导入该模块：

```
import sys
```

导入 `sys` 模块后，我们就有了变量 `sys` 指向该模块，利用 `sys` 这个变量，就可以访问 `sys` 模块的所有功能。

`sys` 模块有一个 `argv` 变量，用list存储了命令行的所有参数。 `argv` 至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行 `python3 hello.py` 获得的 `sys.argv` 就是 `['hello.py']` ；

运行 `python3 hello.py Michael` 获得的 `sys.argv` 就是 `['hello.py', 'Michael]` 。

最后，注意到这两行代码：

```
if __name__=='__main__':
    test()
```

当我们在命令行运行 `hello` 模块文件时，Python解释器把一个特殊变量 `__name__` 置为 `__main__` ，而如果在其他地方导入该 `hello` 模块时， `if` 判断将失败，因此，这种 `if` 测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行 `hello.py` 看看效果：

```
$ python3 hello.py
Hello, world!
$ python hello.py Michael
Hello, Michael!
```

如果启动Python交互环境，再导入 `hello` 模块：

```
$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more informati
>>> import hello
>>>
```

导入时，没有打印`Hello, word!`，因为没有执行`test()`函数。

调用`hello.test()`时，才能打印出`Hello, word!`：

```
>>> hello.test()
Hello, world!
```

作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过`_`前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：`abc`，`x123`，`PI`等；

类似`__xxx__`这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的`__author__`，`__name__`就是特殊变量，`hello`模块定义的文档注释也可以用特殊变量`__doc__`访问，我们自己的变量一般不要用这种变量名；

类似`_xxx`和`__xxx`这样的函数或变量就是非公开的（private），不应该被直接引用，比如`_abc`，`__abc`等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

```
def _private_1(name):
    return 'Hello, %s' % name

def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

我们在模块里公开 `greeting()` 函数，而把内部逻辑用private函数隐藏起来了，这样，调用 `greeting()` 函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

安装第三方模块

在Python中，安装第三方模块，是通过包管理工具pip完成的。

如果你正在使用Mac或Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用Windows，请参考安装Python一节的内容，确保安装时勾选了 `pip` 和 `Add python.exe to Path` 。

在命令提示符窗口下尝试运行 `pip` ，如果Windows提示未找到命令，可以重新运行安装程序添加 `pip` 。

注意：Mac或Linux上有可能并存Python 3.x和Python 2.x，因此对应的pip命令是 `pip3` 。

现在，让我们来安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。不过，PIL目前只支持到Python 2.7，并且有年头没有更新了，因此，基于PIL的Pillow项目开发非常活跃，并且支持最新的Python 3。

一般来说，第三方库都会在Python官方的pypi.python.org网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫Pillow，因此，安装Pillow的命令就是：

```
pip install Pillow
```

耐心等待下载并安装后，就可以使用Pillow了。

有了Pillow，处理图片易如反掌。随便找个图片生成缩略图：

```
>>> from PIL import Image
>>> im = Image.open('test.png')
>>> print(im.format, im.size, im.mode)
PNG (400, 300) RGB
>>> im.thumbnail((200, 100))
>>> im.save('thumb.jpg', 'JPEG')
```

其他常用的第三方库还有MySQL的驱动：`mysql-connector-python`，用于科学计算的NumPy库：`numpy`，用于生成文本的模板工具`Jinja2`，等等。

模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在`sys`模块的`path`变量中：

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.4/lib/python3
```



如果我们要添加自己的搜索目录，有两种方法：

一是直接修改`sys.path`，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量`PYTHONPATH`，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
def print_score(std):
    print('%s: %s' % (std['name'], std['score']))
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 `Student` 这种数据类型应该被视为一个对象，这个对象拥有 `name` 和 `score` 这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 `print_score` 消息，让对象自己把自己的数据打印出来。

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student类为例，在Python中，定义类是通过 `class` 关键字：

```
class Student(object):
    pass
```

`class` 后面紧接着是类名，即 `Student` ，类名通常是大写开头的单词，紧接着是 `(object)` ，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用 `object` 类，这是所有类最终都会继承的类。

定义好了 `Student` 类，就可以根据 `Student` 类创建出 `Student` 的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量 `bart` 指向的就是一个 `Student` 的实例，后面的 `0x10a67a590` 是内存地址，每个object的地址都不一样，而 `Student` 本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例 `bart` 绑定一个 `name` 属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 `__init__` 方法，在创建实例的时候，就把 `name` ， `score` 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意到 `__init__` 方法的第一个参数永远是 `self` ，表示创建的实例本身，因此，在 `__init__` 方法内部，就可以把各种属性绑定到 `self` ，因为 `self` 就指向创建的实例本身。

有了 `__init__` 方法，在创建实例的时候，就不能传入空的参数了，必须传入与 `__init__` 方法匹配的参数，但 `self` 不需要传，Python解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 `self` ，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

数据封装

面向对象编程的一个重要特点就是数据封装。在上面的 `Student` 类中，每个实例就拥有各自的 `name` 和 `score` 这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

```
>>> def print_score(std):
...     print('%s: %s' % (std.name, std.score))
...
>>> print_score(bart)
Bart Simpson: 59
```

但是，既然 `Student` 实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在 `Student` 类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和 `Student` 类本身是关联起来的，我们称之为类的方法：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

要定义一个方法，除了第一个参数是 `self` 外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了 `self` 不用传递，其他参数正常传入：

```
>>> bart.print_score()
Bart Simpson: 59
```

这样一来，我们从外部看 `Student` 类，就只需要知道，创建实例需要给出 `name` 和 `score` ，而如何打印，都是在 `Student` 类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给 `Student` 类增加新的方法，比如 `get_grade` ：

```
class Student(object):
    ...

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同样的，`get_grade`方法可以直接在实例变量上调用，不需要知道内部实现细节：

```
>>> bart.get_grade()
'C'
```

小结

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

```
>>> bart = Student('Bart Simpson', 59)
>>> lisa = Student('Lisa Simpson', 87)
>>> bart.age = 8
>>> bart.age
8
>>> lisa.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

参考源码

student.py

访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的 `name` 、 `score` 属性：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.score
98
>>> bart.score = 59
>>> bart.score
59
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线 `__` ，在Python中，实例的变量名如果以 `__` 开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问 `实例变量.__name` 和 `实例变量.__score` 了：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute '__name'
```

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加 `get_name` 和 `get_score` 这样的方法：

```
class Student(object):
    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score
```

如果又要允许外部代码修改score怎么办？可以再给Student类增加 `set_score` 方法：

```
class Student(object):
    ...

    def set_score(self, score):
        self.__score = score
```

你也许会问，原先那种直接通过 `bart.score = 59` 也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

```
class Student(object):
    ...

    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

需要注意的是，在Python中，变量名类似 `__xxx__` 的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用 `__name__` 、 `__score__` 这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如 `_name` ，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问 `__name` 是因为Python解释器对外把 `__name` 变量改成了 `_Student__name` ，所以，仍然可以通过 `_Student__name` 来访问 `__name` 变量：

```
>>> bart._Student__name
'Bart Simpson'
```

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把 `__name` 改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

参考源码

protected_student.py

继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为 `Animal` 的class，有一个 `run()` 方法可以直接打印：

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

当我们需要编写 `Dog` 和 `Cat` 类时，就可以直接从 `Animal` 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

对于 `Dog` 来说， `Animal` 就是它的父类，对于 `Animal` 来说， `Dog` 就是它的子类。 `Cat` 和 `Dog` 类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于 `Animial` 实现了 `run()` 方法，因此， `Dog` 和 `Cat` 作为它的子类，什么事也没干，就自动拥有了 `run()` 方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如Dog类：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是 `Dog` 还是 `Cat` ，它们 `run()` 的时候，显示的都是 `Animal is running...` ，符合逻辑的做法是分别显示 `Dog is running...` 和 `Cat is running...` ，因此，对 `Dog` 和 `Cat` 类改进如下：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

class Cat(Animal):

    def run(self):
        print('Cat is running...')
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类和父类都存在相同的 `run()` 方法时，我们说，子类的 `run()` 覆盖了父类的 `run()` ，在代码运行的时候，总是会调用子类的 `run()` 。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 `isinstance()` 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 `a` 、 `b` 、 `c` 确实对应着 `list` 、 `Animal` 、 `Dog` 这3种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 `c` 不仅仅是 `Dog` ， `c` 还是 `Animal` ！

不过仔细想想，这是有道理的，因为 `Dog` 是从 `Animal` 继承下来的，当我们创建了一个 `Dog` 的实例 `c` 时，我们认为 `c` 的数据类型是 `Dog` 没错，但 `c` 同时也是 `Animal` 也没错， `Dog` 本来就是 `Animal` 的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

`Dog` 可以看成 `Animal` ，但 `Animal` 不可以看成 `Dog` 。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个 `Animal` 类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 `Animal` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 `Dog` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 `Cat` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个 `Tortoise` 类型，也从 `Animal` 派生：

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')
```

当我们调用 `run_twice()` 时，传入 `Tortoise` 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个 `Animal` 的子类，不必对 `run_twice()` 做任何修改，实际上，任何依赖 `Animal` 作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

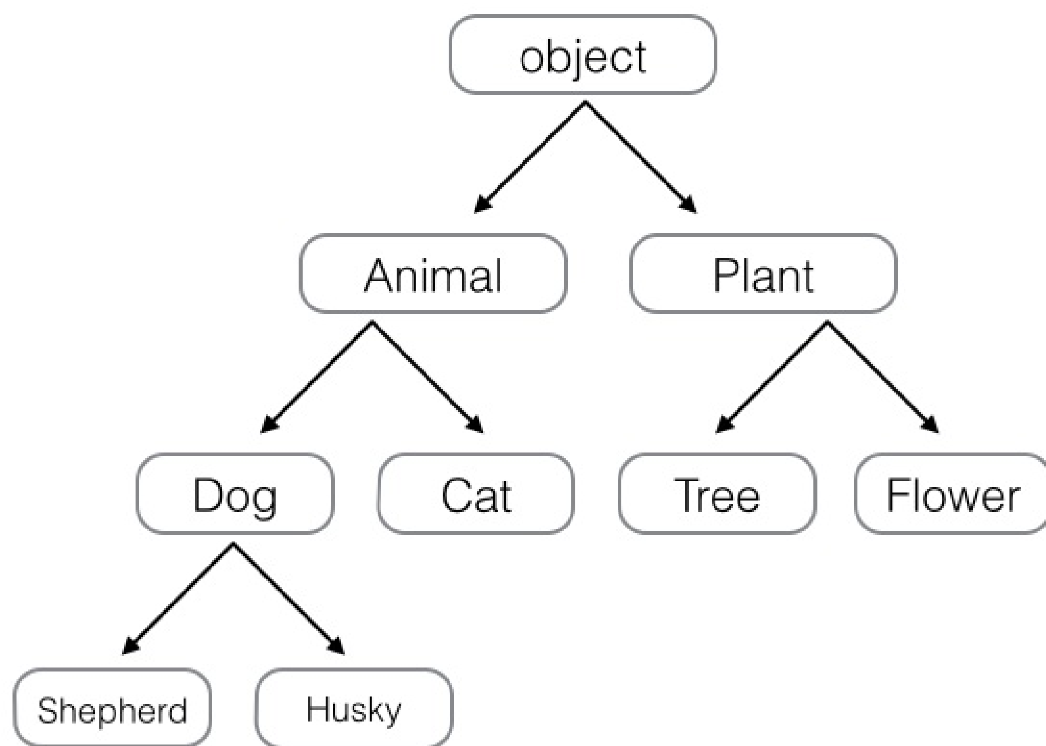
多态的好处就是，当我们需要传入 `Dog` 、 `Cat` 、 `Tortoise` ……时，我们只需要接收 `Animal` 类型就可以了，因为 `Dog` 、 `Cat` 、 `Tortoise` ……都是 `Animal` 类型，然后，按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()` 方法，因此，传入的任意类型，只要是 `Animal` 类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

对于一个变量，我们只需要知道它是 `Animal` 类型，无需确切地知道它的子类型，就可以放心地调用 `run()` 方法，而具体调用的 `run()` 方法是作用在 `Animal` 、 `Dog` 、 `Cat` 还是 `Tortoise` 对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种 `Animal` 的子类时，只要确保 `run()` 方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增 `Animal` 子类；

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



静态语言 vs 动态语言

对于静态语言（例如Java）来说，如果需要传入 `Animal` 类型，则传入的对象必须是 `Animal` 类型或者它的子类，否则，将无法调用 `run()` 方法。

对于Python这样的动态语言来说，则不一定需要传入 `Animal` 类型。我们只需要保证传入的对象有一个 `run()` 方法就可以了：

```
class Timer(object):
    def run(self):
        print('Start...')
```

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个 `read()` 方法，返回其内容。但是，许多对象，只要有 `read()` 方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了 `read()` 方法的对象。

小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的。

参考源码

animals.py

获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用type()

首先，我们来判断对象类型，使用 `type()` 函数：

基本类型都可以用 `type()` 判断：

```
>>> type(123)
<class 'int'>
>>> type('str')
<class 'str'>
>>> type(None)
<type(None) 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 `type()` 判断：

```
>>> type(abs)
<class 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是 `type()` 函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在 `if` 语句中判断，就需要比较两个变量的type类型是否相同：

```
>>> type(123)==type(456)
True
>>> type(123)==int
True
>>> type('abc')==type('123')
True
>>> type('abc')==str
True
>>> type('abc')==type(123)
False
```

判断基本数据类型可以直接写 `int` ， `str` 等，但如果要判断一个对象是否是函数怎么办？可以使用 `types` 模块中定义的常量：

```
>>> import types
>>> def fn():
...     pass
...
>>> type(fn)==types.FunctionType
True
>>> type(abs)==types.BuiltinFunctionType
True
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
True
```

使用isinstance()

对于class的继承关系来说，使用 `type()` 就很不方便。我们要判断class的类型，可以使用 `isinstance()` 函数。

我们回顾上次的例子，如果继承关系是：

```
object -> Animal -> Dog -> Husky
```

那么，`isinstance()`就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为`h`变量指向的就是Husky对象。

再判断：

```
>>> isinstance(h, Dog)
True
```

`h`虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以，`h`也还是Dog类型。换句话说，`isinstance()`判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信，`h`还是Animal类型：

```
>>> isinstance(h, Animal)
True
```

同理，实际类型是Dog的`d`也是Animal类型：

```
>>> isinstance(d, Dog) and isinstance(d, Animal)
True
```

但是，`d`不是Husky类型：

```
>>> isinstance(d, Husky)
False
```

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断：

```
>>> isinstance('a', str)
True
>>> isinstance(123, int)
True
>>> isinstance(b'a', bytes)
True
```

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

```
>>> isinstance([1, 2, 3], (list, tuple))
True
>>> isinstance((1, 2, 3), (list, tuple))
True
```

使用dir()

如果要获得一个对象的所有属性和方法，可以使用 `dir()` 函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
```

类似 `__xxx__` 的属性和方法在Python中都是有特殊用途的，比如 `__len__` 方法返回长度。在Python中，如果你调用 `len()` 函数试图获取一个对象的长度，实际上，在 `len()` 函数内部，它自动去调用该对象的 `__len__()` 方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

我们自己写的类，如果也想用 `len(myObj)` 的话，就自己写一个 `__len__()` 方法：

```
>>> class MyDog(object):
...     def __len__(self):
...         return 100
...
>>> dog = MyDog()
>>> len(dog)
100
```

剩下的都是普通属性或方法，比如 `lower()` 返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

仅仅把属性和方法列出来是不够的，配合 `getattr()` 、 `setattr()` 以及 `hasattr()` ，我们可以直接操作一个对象的状态：

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗？
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗？
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗？
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出AttributeError的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个default参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404
404
```

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗？
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x1007
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
>>> fn # fn指向obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x1007
>>> fn() # 调用fn()与调用obj.power()是一样的
81
```

小结

通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

```
sum = obj.x + obj.y
```

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。 `hasattr()` 就派上了用场。

请注意，在Python这类动态语言中，根据鸭子类型，有 `read()` 方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要 `read()` 方法返回的是有效的图像数据，就不影响读取图像的功能。

参考源码

get_type.py

attrs.py

实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过 `self` 变量：

```
class Student(object):
    def __init__(self, name):
        self.name = name

s = Student('Bob')
s.score = 90
```

但是，如果 `Student` 类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归 `Student` 类所有：

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

```
>>> class Student(object):
...     name = 'Student'
...
>>> s = Student() # 创建实例s
>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找cl
Student
>>> print(Student.name) # 打印类的name属性
Student
>>> s.name = 'Michael' # 给实例绑定name属性
>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属
Michael
>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问
Student
>>> del s.name # 如果删除实例的name属性
>>> print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属
Student
```

从上面的例子可以看出，在编写程序的时候，千万不要把实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

使用__slots__

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

```
class Student(object):
    pass
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给class绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = MethodType(set_score, Student)
```

给class绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的 `set_score` 方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

使用slots

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加 `name` 和 `age` 属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的 `__slots__` 变量，来限制该class实例能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于 `'score'` 没有被放到 `__slots__` 中，所以不能绑定 `score` 属性，试图绑定 `score` 将得到 `AttributeError` 的错误。

使用 `__slots__` 要注意， `__slots__` 定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义 `__slots__` ，这样，子类实例允许定义的属性就是自身的 `__slots__` 加上父类的 `__slots__` 。

参考源码

use_slots.py

使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()
s.score = 9999
```

这显然不合逻辑。为了限制score的范围，可以通过一个 `set_score()` 方法来设置成绩，再通过一个 `get_score()` 来获取成绩，这样，在 `set_score()` 方法里，就可以检查参数：

```
class Student(object):

    def get_score(self):
         return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的 `@property` 装饰器就是负责把一个方法变成属性调用的：

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

`@property` 的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上 `@property` 就可以了，此时， `@property` 本身又创建了另一个装饰器 `@score.setter` ，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK，实际转化为s.set_score(60)
>>> s.score # OK，实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的 `@property` ，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

上面的 `birth` 是可读写属性，而 `age` 就是一个只读属性，因为 `age` 可以根据 `birth` 和当前时间计算出来。

小结

`@property` 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

练习

请利用 `@property` 给一个 `Screen` 对象加上 `width` 和 `height` 属性，以及一个只读属性 `resolution` ：

```
# -*- coding: utf-8 -*-

class Screen(object):

    pass

# test:
s = Screen()
s.width = 1024
s.height = 768
print(s.resolution)
assert s.resolution == 786432, '1024 * 768 = %d ?' % s.resolution
```

参考源码

use_property.py

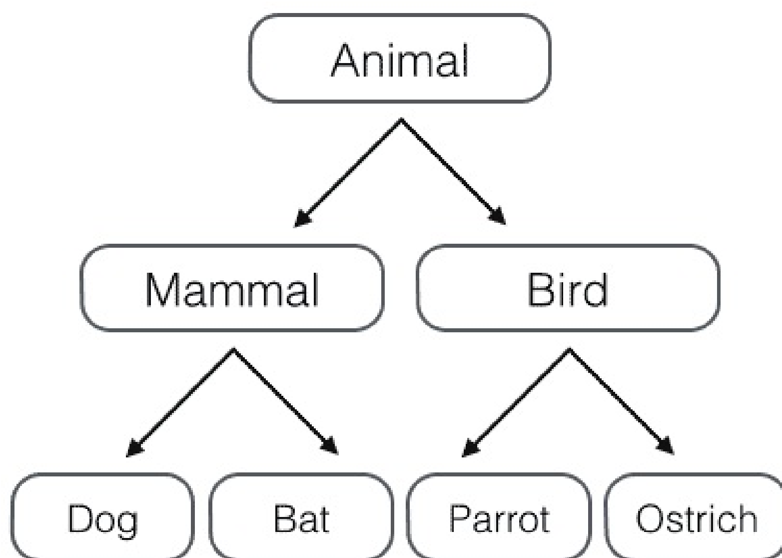
多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

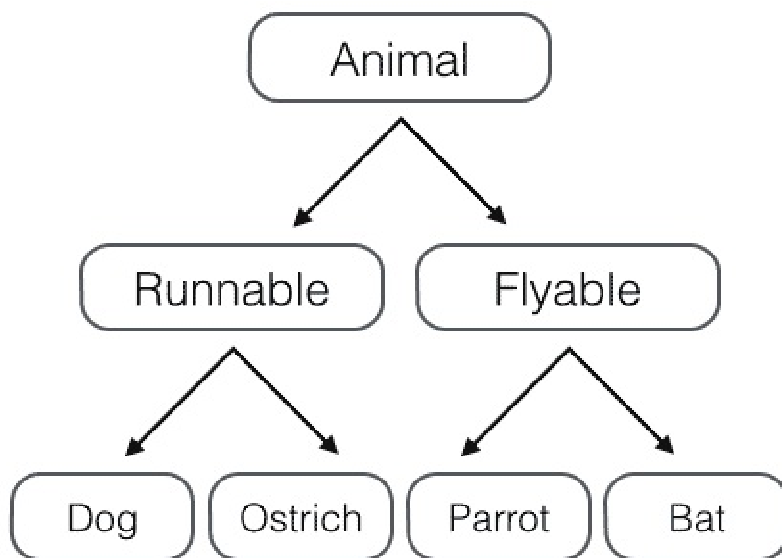
回忆一下 `Animal` 类层次的设计，假设我们要实现以下4种动物：

- Dog - 狗狗；
- Bat - 蝙蝠；
- Parrot - 鹦鹉；
- Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



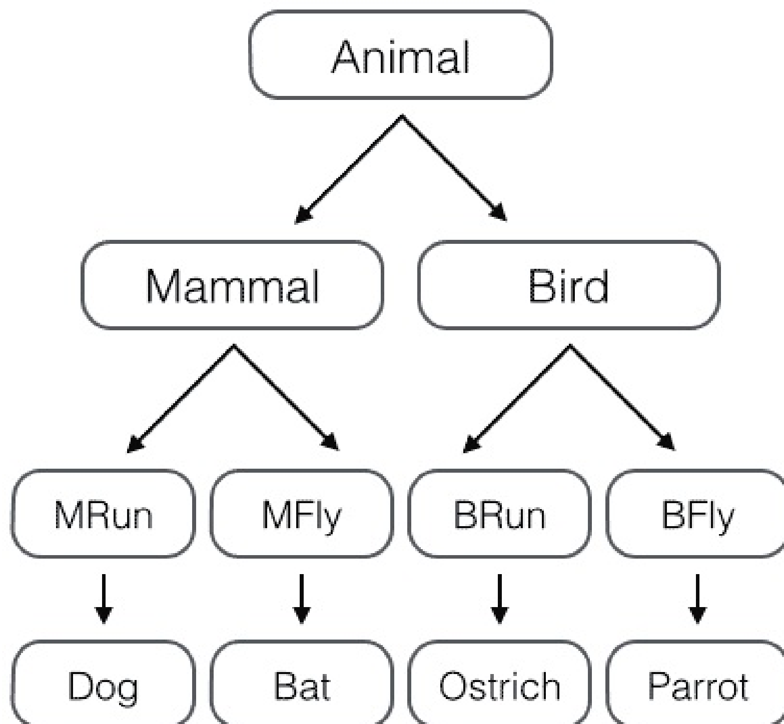
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

- 哺乳类：能跑的哺乳类，能飞的哺乳类；
- 鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

```
class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass

class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass

class Bat(Mammal):
    pass

class Parrot(Bird):
    pass

class Ostrich(Bird):
    pass
```

现在，我们要给动物再加上 `Runnable` 和 `Flyable` 的功能，只需要先定义好 `Runnable` 和 `Flyable` 的类：

```
class Runnable(object):
    def run(self):
        print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')
```

对于需要 `Runnable` 功能的动物，就多继承一个 `Runnable` ，例如 `Dog` ：

```
class Dog(Mammal, Runnable):
    pass
```

对于需要 `Flyable` 功能的动物，就多继承一个 `Flyable` ，例如 `Bat` ：

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如， `Ostrich` 继承自 `Bird` 。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让 `Ostrich` 除了继承自 `Bird` 外，再同时继承 `Runnable` 。这种设计通常称之为MixIn。

为了更好地看出继承关系，我们把 `Runnable` 和 `Flyable` 改为 `RunnableMixIn` 和 `FlyableMixIn` 。类似的，你还可以定义出肉食动物 `CarnivorousMixIn` 和植食动物 `HerbivoresMixIn` ，让某个动物同时拥有好几个MixIn：

```
class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):
    pass
```

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了MixIn。举个例子，Python自带了 `TCPServer` 和 `UDPServer` 这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由 `ForkingMixIn` 和 `ThreadingMixIn` 提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

```
class MyTCPServer(TCPServer, ForkingMixIn):
    pass
```

编写一个多线程模式的UDP服务，定义如下：

```
class MyUDPServer(UDPServer, ThreadingMixIn):
    pass
```

如果你打算搞一个更先进的协程模型，可以编写一个`CoroutineMixIn`：

```
class MyTCPServer(TCPServer, CoroutineMixIn):
    pass
```

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

小结

由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用MixIn的设计。

定制类

看到类似 `__slots__` 这种形如 `__xxx__` 的变量或者函数名就要注意，这些在Python中是有特殊用途的。

`__slots__` 我们已经知道怎么用了， `__len__()` 方法我们也知道是为了能让class作用于 `len()` 函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

str

我们先定义一个 `Student` 类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

打印出一堆 `<__main__.Student object at 0x109afb190>` ，不好看。

怎么才能打印得好看呢？只需要定义好 `__str__()` 方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用 `print` ，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是 `__str__()` ，而是 `__repr__()` ，两者的区别是 `__str__()` 返回用户看到的字符串，而 `__repr__()` 返回程序开发者看到的字符串，也就是说， `__repr__()` 是为调试服务的。

解决办法是再定义一个 `__repr__()` 。但是通常 `__str__()` 和 `__repr__()` 代码都是一样的，所以，有个偷懒的写法：

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

iter

如果一个类想被用于 `for ... in` 循环，类似list或tuple那样，就必须实现一个 `__iter__()` 方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的 `__next__()` 方法拿到循环的下一个值，直到遇到 `StopIteration` 错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration();
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
...
46368
75025
```

getitem

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

```
>>> Fib()[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Fib' object does not support indexing
```

要表现得像list那样按照下标取出元素，需要实现 `__getitem__()` 方法：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

但是list有个神奇的切片方法：

```
>>> list(range(100))[5:10]
[5, 6, 7, 8, 9]
```

对于Fib却报错。原因是 `__getitem__()` 传入的参数可能是一个int，也可能是一个切片对象 `slice` ，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

现在试试Fib的切片：

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理：

```
>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

也没有对负数作处理，所以，要正确实现一个 `__getitem__()` 还是有很多工作要做的。

此外，如果把对象看成 `dict` ， `__getitem__()` 的参数也可能是一个可以作key的object，例如 `str` 。

与之对应的是 `__setitem__()` 方法，把对象视作list或dict来对集合赋值。最后，还有一个 `__delitem__()` 方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

getattr

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义 `Student` 类：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'
```

调用 `name` 属性，没问题，但是，调用不存在的 `score` 属性，就有问题了：

```
>>> s = Student()
>>> print(s.name)
Michael
>>> print(s.score)
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'score'
```

错误信息很清楚地告诉我们，没有找到 `score` 这个attribute。

要避免这个错误，除了可以加上一个 `score` 属性外，Python还有另一个机制，那就是写一个 `__getattr__()` 方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如 `score` ，Python解释器会试图调用 `__getattr__(self, 'score')` 来尝试获得属性，这样，我们就有机会返回 `score` 的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用 `__getattr__` ，已有的属性，比如 `name` ，不会在 `__getattr__` 中查找。

此外，注意到任意调用如 `s.abc` 都会返回 `None` ，这是因为我们定义的 `__getattr__` 默认返回就是 `None` 。要让class只响应特定的几个属性，我们就要按照约定，抛出 `AttributeError` 的错误：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no attribute \
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

- http://api.server/user/friends
- http://api.server/user/timeline/list

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的 `__getattr__` ，我们可以写出一个链式调用：

```
class Chain(object):

    def __init__(self, path=''):
        self._path = path

    def __getattr__(self, path):
        return Chain('%s/%s' % (self._path, path))

    def __str__(self):
        return self._path

    __repr__ = __str__
```

试试：

```
>>> Chain().status.user.timeline.list
'/status/user/timeline/list'
```

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

```
GET /users/:user/repos
```

调用时，需要把 `:user` 替换为实际用户名。如果我们能写出这样的链式调用：

```
Chain().users('michael').repos
```

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

call

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用 `instance.method()` 来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个 `__call__()` 方法，就可以直接对实例进行调用。请看示例：

```
class Student(object):
    def __init__(self, name):
        self.name = name

    def __call__(self):
        print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

`__call__()` 还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个 `Callable` 对象，比如函数和我们上面定义的带有 `__call__()` 的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('str')
False
```

通过 `callable()` 函数，我们就可以判断一个对象是否是“可调用”对象。

小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考Python的官方文档。

参考源码

special_str.py

special_iter.py

special_getitem.py

special_getattr.py

special_call.py

使用枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

```
JAN = 1
FEB = 2
MAR = 3
...
NOV = 11
DEC = 12
```

好处是简单，缺点是类型是 `int` ，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了 `Enum` 类来实现这个功能：

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'J
```

这样我们就获得了 `Month` 类型的枚举类，可以直接使用 `Month.Jan` 来引用一个常量，或者枚举它的所有成员：

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

`value` 属性则是自动赋给成员的 `int` 常量，默认从 `1` 开始计数。

如果需要更精确地控制枚举类型，可以从 `Enum` 派生出自定义类：

```
from enum import Enum, unique

@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

`@unique` 装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
>>> day1 = Weekday.Mon
>>> print(day1)
Weekday.Mon
>>> print(Weekday.Tue)
Weekday.Tue
>>> print(Weekday['Tue'])
Weekday.Tue
>>> print(Weekday.Tue.value)
2
>>> print(day1 == Weekday.Mon)
True
>>> print(day1 == Weekday.Tue)
False
>>> print(Weekday(1))
Weekday.Mon
>>> print(day1 == Weekday(1))
True
>>> Weekday(7)
Traceback (most recent call last):
  ...
ValueError: 7 is not a valid Weekday
>>> for name, member in Weekday.__members__.items():
...     print(name, '=>', member)
...
Sun => Weekday.Sun
Mon => Weekday.Mon
Tue => Weekday.Tue
Wed => Weekday.Wed
Thu => Weekday.Thu
Fri => Weekday.Fri
Sat => Weekday.Sat
```

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

小结

`Enum` 可以把一组相关常量定义在一个class中，且class不可变，而且成员可以直接比较。

参考源码

use_enum.py

使用元类

type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个 `Hello` 的class，就写一个 `hello.py` 模块：

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

当Python解释器载入 `hello` 模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个 `Hello` 的class对象，测试如下：

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

`type()` 函数可以查看一个类型或变量的类型， `Hello` 是一个class，它的类型就是 `type` ，而 `h` 是一个实例，它的类型就是class `Hello` 。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用 `type()` 函数。

`type()` 函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过 `type()` 函数创建出 `Hello` 类，而无需通过 `class Hello(object)...` 的定义：

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello cl
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

要创建一个class对象，`type()`函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数`fn`绑定到方法名`hello`上。

通过`type()`函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class。

正常情况下，我们都用`class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

metaclass

除了使用`type()`动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个 `add` 方法：

定义 `ListMetaclass` ，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

```
# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数 `metaclass` ：

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

当我们传入关键字参数 `metaclass` 时，魔术就生效了，它指示Python解释器在创建 `MyList` 时，要通过 `ListMetaclass.__new__()` 来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

`__new__()` 方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；

3. 类继承的父类集合；

4. 类的方法集合。

测试一下`MyList`是否可以调用`add()`方法：

```
>>> L = MyList()
>>> L.add(1)
>> L
[1]
```

而普通的`list`没有`add()`方法：

```
>>> L2 = list()
>>> L2.add(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'
```

动态修改有什么意义？直接在`MyList`定义中写上`add()`方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个`User`类来操作对应的数据库表`User`，我们期待他写出这样的代码：

```
class User(Model):
    # 定义类的属性到列的映射：
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例：
u = User(id=12345, name='Michael', email='test@orm.org', password=
# 保存到数据库：
u.save()
```

其中，父类 `Model` 和属性类型 `StringField` 、 `IntegerField` 是由ORM框架提供的，剩下的魔术方法比如 `save()` 全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义 `Field` 类，它负责保存数据库表的字段名和字段类型：

```
class Field(object):

    def __init__(self, name, column_type):
        self.name = name
        self.column_type = column_type

    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__, self.name)
```

在 `Field` 的基础上，进一步定义各种类型的 `Field` ，比如 `StringField` ， `IntegerField` 等等：

```
class StringField(Field):

    def __init__(self, name):
        super(StringField, self).__init__(name, 'varchar(100)')

class IntegerField(Field):

    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')
```

下一步，就是编写最复杂的 `ModelMetaclass` 了：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        print('Found model: %s' % name)
        mappings = dict()
        for k, v in attrs.items():
            if isinstance(v, Field):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
        for k in mappings.keys():
            attrs.pop(k)
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = name # 假设表名和类名一致
        return type.__new__(cls, name, bases, attrs)
```

以及基类 `Model` ：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__,
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

当用户定义一个 `class User(Model)` 时，Python解释器首先在当前类 `User` 的定义中查找 `metaclass` ，如果没有找到，就继续在父类 `Model` 中查找 `metaclass` ，找到了，就使用 `Model` 中定义的 `metaclass` 的 `ModelMetaclass` 来创建 `User` 类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在 `ModelMetaclass` 中，一共做了几件事情：

1. 排除掉对 `Model` 类的修改；

2. 在当前类（比如 `User` ）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个 `__mappings__` 的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；
3. 把表名保存到 `__table__` 中，这里简化为表名默认为类名。

在 `Model` 类中，就可以定义各种操作数据库的方法，比如 `save()` ， `delete()` ， `find()` ， `update` 等等。

我们实现了 `save()` 方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出 `INSERT` 语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org', password=
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,id) values (?,?,?,?)
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
```

可以看到， `save()` 方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架。

小结

metaclass是Python中非常具有魔术性的对象，它可以改变类创建时的行为。这种强大的功能使用起来务必小心。

参考源码

create_class_on_the_fly.py

use_metaclass.py

orm.py

错误、调试和测试

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数 `open()` ，成功时返回文件描述符（就是一个整数），出错时返回 `-1` 。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print('Error')
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python也不例外。

try

让我们用一个例子来看看 `try` 的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块，至此，执行完毕。

上面的代码在计算 `10 / 0` 时会产生一个除法运算错误：

```
try...
except: division by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句 `print('result:', r)` 不会被执行， `except` 由于捕获到 `ZeroDivisionError` ，因此被执行。最后， `finally` 语句被执行。然后，程序继续按照流程往下走。

如果把除数 `0` 改成 `2` ，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以 `except` 语句块不会被执行，但是 `finally` 如果有，则一定会被执行（可以没有 `finally` 语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 `except` 语句块处理。没错，可以有多个 `except` 来捕获不同类型的错误：

```
try:
    print('try...')
    r = 10 / int('a')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

`int()` 函数可能会抛出 `ValueError` ，所以我们用一个 `except` 捕获 `ValueError` ，用另一个 `except` 捕获 `ZeroDivisionError` 。

此外，如果没有错误发生，可以在 `except` 语句块后面加一个 `else` ，当没有错误发生时，会自动执行 `else` 语句：

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python的错误其实也是class，所有的错误类型都继承自 `BaseException` ，所以在使用 `except` 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

第二个 `except` 永远也捕获不到 `UnicodeError` ，因为 `UnicodeError` 是 `ValueError` 的子类，如果有，也被第一个 `except` 给捕获了。

Python所有的错误都是从 `BaseException` 类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

使用 `try...except` 捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数 `main()` 调用 `foo()` ， `foo()` 调用 `bar()` ，结果 `bar()` 出错了，这时，只要 `main()` 捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 `try...except...finally` 的麻烦。

调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看 `err.py` ：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2~3行：

```
  File "err.py", line 11, in <module>
    main()
```

调用 `main()` 出错了，在代码文件 `err.py` 的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用 `bar('0')` 出错了，在代码文件 `err.py` 的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是 `return foo(s) * 2` 这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是 `return 10 / int(s)` 这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型 `ZeroDivisionError`，我们判断，`int(s)` 本身并没有出错，但是 `int(s)` 返回 `0`，在计算 `10 / 0` 时出错，至此，找到错误源头。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的 `logging` 模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)

main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
  File "err_logging.py", line 13, in main
    bar('0')
  File "err_logging.py", line 9, in bar
    return foo(s) * 2
  File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置， `logging` 还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用 `raise` 语句抛出一个错误的实例：

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n

foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如 `ValueError` ， `TypeError` ），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

在 `bar()` 函数中，我们明明已经捕获了错误，但是，打印一个 `ValueError!` 后，又把错误通过 `raise` 语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

`raise` 语句如果不带参数，就会把当前错误原样抛出。此外，在 `except` 中 `raise` 一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 `IOError` 转换成毫不相干的 `ValueError` 。

小结

Python内置的 `try...except...finally` 用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

参考源码

do_try.py

err.py

err_logging.py

err_raise.py

err_reraise.py

调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用 `print()` 把可能有问题的变量打印出来看看：

```
def foo(s):
    n = int(s)
    print('>>> n = %d' % n)
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python3 err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用 `print()` 最大的坏处是将来还得删掉它，想想程序里到处都是 `print()` ，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用 `print()` 来辅助查看的地方，都可以用断言（assert）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert` 的意思是，表达式 `n != 0` 应该是 `True` ，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败， `assert` 语句本身就会抛出 `AssertionError` ：

```
$ python3 err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着 `assert` ，和 `print()` 相比也好不到哪去。不过，启动Python解释器时可以用 `-O` 参数来关闭 `assert` ：

```
$ python3 -O err.py
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

关闭后，你可以把所有的 `assert` 语句当成 `pass` 来看。

logging

把 `print()` 替换为 `logging` 是第3种方式，和 `assert` 比， `logging` 不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

`logging.info()`就可以输出一段文本。运行，发现除了`ZeroDivisionError`，没有任何信息。怎么回事？

别急，在`import logging`之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python3 err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是`logging`的好处，它允许你指定记录信息的级别，有`debug`，`info`，`warning`，`error`等几个级别，当我们指定`level=INFO`时，`logging.debug`就不起作用了。同理，指定`level=WARNING`后，`debug`和`info`就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging`的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python3 -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<modul
-> s = '0'
```

以参数 `-m pdb` 启动后，pdb定位到下一步要执行的代码 `-> s = '0'` 。输入命令 `l` 来查看代码：

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print(10 / n)
```

输入命令 `n` 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<modul
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<modul
-> print(10 / n)
```

任何时候都可以输入命令 `p 变量名` 来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令 `q` 结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

pdb.set_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要 `import pdb` ，然后，在可能出错的地方放一个 `pdb.set_trace()` ，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在 `pdb.set_trace()` 暂停并进入pdb调试环境，可以用命令 `p` 查看变量，或者用命令 `c` 继续运行：

```
$ python3 err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<modul
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有PyCharm：

http://www.jetbrains.com/pycharm/

另外，Eclipse加上pydev插件也可以调试Python程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

参考源码

do_assert.py

do_logging.py

do_pdb.py

单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数 `abs()` ，我们可以编写出以下几个测试用例：

1. 输入正数，比如 `1` 、 `1.2` 、 `0.99` ，期待返回值与输入相同；
2. 输入负数，比如 `-1` 、 `-1.2` 、 `-0.99` ，期待返回值与输入相反；
3. 输入 `0` ，期待返回 `0` ；
4. 输入非数值类型，比如 `None` 、 `[]` 、 `{}` ，期待抛出 `TypeError` 。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对 `abs()` 函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对 `abs()` 函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个 `Dict` 类，这个类的行为和 `dict` 一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py` 代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的 `unittest` 模块，编写 `mydict_test.py` 如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')

    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEqual()` ：

```
self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过 `d['empty']` 访问不存在的key时，断言会抛出 `KeyError` ：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过 `d.empty` 访问不存在的key时，我们期待抛出 `AttributeError` ：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在 `mydict_test.py` 的最后加上两行代码：

```
if __name__ == '__main__':
    unittest.main()
```

这样就可以把 `mydict_test.py` 当做正常的python脚本运行：

```
$ python3 mydict_test.py
```

另一种方法是在命令行通过参数 `-m unittest` 直接运行单元测试：

```
$ python3 -m unittest mydict_test
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的 `setUp()` 和 `tearDown()` 方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()` 和 `tearDown()` 方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 `setUp()` 方法中连接数据库，在 `tearDown()` 方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

可以再次运行测试看看每个测试方法调用前后是否会打印出 `setUp...` 和 `tearDown...` 。

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

参考源码

mydict.py

mydict_test.py

文档测试

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如re模块就带了很多示例代码：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用 `...` 表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的 `Dict` 类：

```
# mydict2.py
class Dict(dict):
    '''
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
        ...
    KeyError: 'empty'
    >>> d2.empty
    Traceback (most recent call last):
        ...
    AttributeError: 'Dict' object has no attribute 'empty'
    '''
    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute
```

```
    def __setattr__(self, key, value):
        self[key] = value

if __name__=='__main__':
    import doctest
    doctest.testmod()
```

运行 `python3 mydict2.py` ：

```
$ python3 mydict2.py
```

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把 `__getattr__()` 方法注释掉，再运行就会报错：

```
$ python3 mydict2.py
**********************************************************************
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py'
Failed example:
    d1.x
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'x'
**********************************************************************
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py'
Failed example:
    d2.c
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'c'
**********************************************************************
1 items had failures:
   2 of   9 in __main__.Dict
***Test Failed*** 2 failures.
```

注意到最后3行代码。当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

练习

对函数 `fact(n)` 编写doctest并执行：

```
# -*- coding: utf-8 -*-

def fact(n):
    '''



    '''
    if n < 1:
        raise ValueError()
    if n == 1:
        return 1
    return n * fact(n - 1)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

小结

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest。

参考源码

mydict2.py

IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在， `open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/Users/mic
```

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 `IOError` ，一旦出错，后面的 `f.close()` 就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

```
try:
    f = open('/path/to/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

但是每次都这么写实在太繁琐，所以，Python引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```
with open('/path/to/file', 'r') as f:
    print(f.read())
```

这和前面的 `try ... finally` 是一样的，但是代码更佳简洁，并且不必调用 `f.close()` 方法。

调用 `read()` 会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取size个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list` 。因此，要根据需要决定怎么调用。

如果文件很小， `read()` 一次性读取最方便；如果不能确定文件大小，反复调用 `read(size)` 比较保险；如果是配置文件，调用 `readlines()` 最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像 `open()` 函数返回的这种有个 `read()` 方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个 `read()` 方法就行。

`StringIO` 就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用 `'rb'` 模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

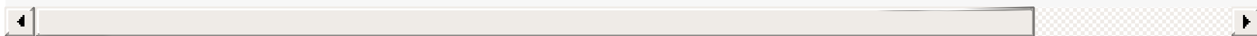
字符编码

要读取非UTF-8编码的文本文件，需要给 `open()` 函数传入 `encoding` 参数，例如，读取GBK编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')
>>> f.read()
'测试'
```

遇到有些编码不规范的文件，你可能会遇到 `UnicodeDecodeError` ，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况， `open()` 函数还接收一个 `errors` 参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors=
```



写文件

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，传入标识符 `'w'` 或者 `'wb'` 表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用 `write()` 来写入文件，但是务必要调用 `f.close()` 来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 `close()` 方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用 `with` 语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请给 `open()` 函数传入 `encoding` 参数，将字符串自动转换成指定编码。

小结

在Python中，文件读写是通过 `open()` 函数打开的文件对象完成的。使用 `with` 语句操作文件IO是个好习惯。

参考源码

with_file.py

StringIO和BytesIO

StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

StringIO顾名思义就是在内存中读写str。

要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可：

```
>>> from io import StringIO
>>> f = StringIO()
>>> f.write('hello')
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue())
hello world!
```

`getvalue()` 方法用于获得写入后的str。

要读取StringIO，可以用一个str初始化StringIO，然后，像读文件一样读取：

```
>>> from io import StringIO
>>> f = StringIO('Hello!\nHi!\nGoodbye!')
>>> while True:
...     s = f.readline()
...     if s == '':
...         break
...     print(s.strip())
...
Hello!
Hi!
Goodbye!
```

BytesIO

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes：

```
>>> from io import BytesIO
>>> f = BytesIO()
>>> f.write('中文'.encode('utf-8'))
6
>>> print(f.getvalue())
b'\xe4\xb8\xad\xe6\x96\x87'
```

请注意，写入的不是str，而是经过UTF-8编码的bytes。

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取：

```
>>> from io import StringIO
>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')
>>> f.read()
b'\xe4\xb8\xad\xe6\x96\x87'
```

小结

StringIO和BytesIO是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

参考源码

do_stringio.py

do_bytesio.py

操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如 `dir` 、 `cp` 等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的 `os` 模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用 `os` 模块的基本功能：

```
>>> import os
>>> os.name # 操作系统类型
'posix'
```

如果是 `posix` ，说明系统是 `Linux` 、 `Unix` 或 `Mac OS X` ，如果是 `nt` ，就是 `Windows` 系统。

要获取详细的系统信息，可以调用 `uname()` 函数：

```
>>> os.uname()
posix.uname_result(sysname='Darwin', nodename='MichaelMacPro.local
```

注意 `uname()` 函数在Windows上不提供，也就是说， `os` 模块的某些函数是跟操作系统相关的。

环境变量

在操作系统中定义的环境变量，全部保存在 `os.environ` 这个变量中，可以直接查看：

```
>>> os.environ
environ({'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'TERM_PROGRAM_VERS
```

要获取某个环境变量的值，可以调用 `os.environ.get('key')` ：

```
>>> os.environ.get('PATH')
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/loc
>>> os.environ.get('x', 'default')
'default'
```

操作文件和目录

操作文件和目录的函数一部分放在 `os` 模块中，一部分放在 `os.path` 模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

```
# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Users/michael'
# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:
>>> os.path.join('/Users/michael', 'testdir')
'/Users/michael/testdir'
# 然后创建一个目录:
>>> os.mkdir('/Users/michael/testdir')
# 删掉一个目录:
>>> os.rmdir('/Users/michael/testdir')
```

把两个路径合成一个时，不要直接拼字符串，而要通过 `os.path.join()` 函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下， `os.path.join()` 返回这样的字符串：

```
part-1/part-2
```

而Windows下会返回这样的字符串：

```
part-1\part-2
```

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过 `os.path.split()` 函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

`os.path.splitext()` 可以直接让你得到文件扩展名，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个 `test.txt` 文件：

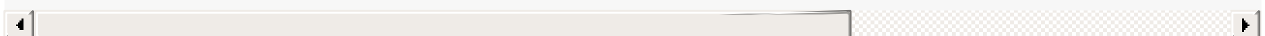
```
# 对文件重命名:
>>> os.rename('test.txt', 'test.py')
# 删掉文件:
>>> os.remove('test.py')
```

但是复制文件的函数居然在 `os` 模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是 `shutil` 模块提供了 `copyfile()` 的函数，你还可以在 `shutil` 模块中找到很多实用函数，它们可以看做是 `os` 模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Appl
```



要列出所有的 `.py` 文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.sp
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py',
```

是不是非常简洁？

小结

Python的 `os` 模块封装了操作系统的目录和文件操作，要注意这些函数有的在 `os` 模块中，有的在 `os.path` 模块中。

练习

1. 利用 `os` 模块编写一个能实现 `dir -l` 输出的程序。
2. 编写一个程序，能在当前目录以及当前目录的所有子目录下查找文件名包含指定字符串的文件，并打印出相对路径。

参考源码

do_dir

序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把 `name` 改成 `'Bill'` ，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的 `'Bill'` 存储到磁盘上，下次重新运行程序，变量又被初始化为 `'Bob'` 。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了 `pickle` 模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

```
>>> import pickle
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00sco
```

`pickle.dumps()` 方法把任意对象序列化成一个 `bytes` ，然后，就可以把这个 `bytes` 写入文件。或者用另一个方法 `pickle.dump()` 直接把对象序列化后写入一个file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 `dump.txt` 文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个 `bytes` ，然后用 `pickle.loads()` 方法反序列化出对象，也可以直接用 `pickle.load()` 方法从一个 `file-like Object` 中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	Python类型
{}	dict
[]	list
"string"	str
1234.56	int或float
true/false	True/False
null	None

Python内置的 `json` 模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`dumps()` 方法返回一个 `str` ，内容就是标准的JSON。类似的， `dump()` 方法可以直接把JSON写入一个 `file-like Object` 。

要把JSON反序列化为Python对象，用 `loads()` 或者对应的 `load()` 方法，前者把JSON的字符串反序列化，后者从 `file-like Object` 中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的 `str` 与JSON的字符串之间转换。

JSON进阶

Python的 `dict` 对象可以直接序列化为JSON的 `{}` ，不过，很多时候，我们更喜欢用 `class` 表示对象，比如定义 `Student` 类，然后序列化：

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 `TypeError` ：

```
Traceback (most recent call last):
  ...
TypeError: <__main__.Student object at 0x10603cc50> is not JSON ser
```

错误的原因是 `Student` 对象不是一个可序列化为JSON的对象。

如果连 `class` 的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看 `dumps()` 方法的参数列表，可以发现，除了第一个必须的 `obj` 参数外， `dumps()` 方法还提供了一大堆的可选参数：

https://docs.python.org/3/library/json.html#json.dumps

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把 `Student` 类实例序列化为JSON，是因为默认情况下， `dumps()` 方法不知道如何将 `Student` 实例变为一个JSON的 `{}` 对象。

可选参数 `default` 就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为 `Student` 专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }
```

这样，`Student` 实例首先被 `student2dict()` 函数转换成 `dict`，然后再被顺利序列化为JSON：

```
>>> print(json.dumps(s, default=student2dict))
{"age": 20, "name": "Bob", "score": 88}
```

不过，下次如果遇到一个 `Teacher` 类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意 `class` 的实例变为 `dict`：

```
print(json.dumps(s, default=lambda obj: obj.__dict__))
```

因为通常 `class` 的实例都有一个 `__dict__` 属性，它就是一个 `dict`，用来存储实例变量。也有少数例外，比如定义了 `__slots__` 的class。

同样的道理，如果我们要把JSON反序列化为一个 `Student` 对象实例，`loads()` 方法首先转换出一个 `dict` 对象，然后，我们传入的 `object_hook` 函数负责把 `dict` 转换为 `Student` 实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])
```

运行结果如下：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> print(json.loads(json_str, object_hook=dict2student))
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 `Student` 实例对象。

小结

Python语言特定的序列化模块是 `pickle` ，但如果要把序列化搞得更通用、更符合Web标准，就可以使用 `json` 模块。

`json` 模块的 `dumps()` 和 `loads()` 函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

参考源码

use_pickle.py

use_json.py

进程和线程

很多同学都听说过，现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核CPU才可能实现。

我们前面编写的所有的Python程序，都是执行单任务的进程，也就是只有一个线程。如果我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务1必须暂停等待任务2完成后才能继续执行，有时，任务3和任务4又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

小结

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个 `fork()` 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是 `fork()` 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回 `0` ，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用 `getppid()` 就可以拿到父进程的ID。

Python的 `os` 模块封装了常见的系统调用，其中就包括 `fork` ，可以在Python程序中轻松创建子进程：

```
import os

print('Process (%s) start...' % os.getpid())
# Only works on Unix/Linux/Mac:
pid = os.fork()
if pid == 0:
    print('I am child process (%s) and my parent is %s.' % (os.getp
else:
    print('I (%s) just created a child process (%s).' % (os.getpid(
```

运行结果如下：

```
Process (876) start...
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

由于Windows没有 `fork` 调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了 `fork` 调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就fork出子进程来处理新的http请求。

multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于Windows没有 `fork` 调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。 `multiprocessing` 模块就是跨平台版本的多进程模块。

`multiprocessing` 模块提供了一个 `Process` 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print('Run child process %s (%s)...' % (name, os.getpid()))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=('test',))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
Parent process 928.
Process will start.
Run child process test (929)...
Process end.
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 `Process` 实例，用 `start()` 方法启动，这样创建进程比 `fork()` 还要简单。

`join()` 方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

执行结果如下：

```
Parent process 669.
Waiting for all subprocesses done...
Run task 0 (671)...
Run task 1 (672)...
Run task 2 (673)...
Run task 3 (674)...
Task 2 runs 0.14 seconds.
Run task 4 (673)...
Task 1 runs 0.27 seconds.
Task 3 runs 0.86 seconds.
Task 0 runs 1.41 seconds.
Task 4 runs 1.91 seconds.
All subprocesses done.
```

代码解读：

对 `Pool` 对象调用 `join()` 方法会等待所有子进程执行完毕，调用 `join()` 之前必须先调用 `close()` ，调用 `close()` 之后就不能继续添加新的 `Process` 了。

请注意输出的结果，task `0` ， `1` ， `2` ， `3` 是立刻执行的，而task `4` 要等待前面某个task完成后才执行，这是因为 `Pool` 的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是 `Pool` 有意设计的限制，并不是操作系统的限制。如果改成：

```
p = Pool(5)
```

就可以同时跑5个进程。

由于 `Pool` 的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

`subprocess` 模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令 `nslookup www.python.org` ，这和命令行直接运行的效果是一样的：

```
import subprocess

print('$ nslookup www.python.org')
r = subprocess.call(['nslookup', 'www.python.org'])
print('Exit code:', r)
```

运行结果：

```
$ nslookup www.python.org
Server:		192.168.19.4
Address:	192.168.19.4#53

Non-authoritative answer:
www.python.org	canonical name = python.map.fastly.net.
Name:	python.map.fastly.net
Address: 199.27.79.223

Exit code: 0
```

如果子进程还需要输入，则可以通过 `communicate()` 方法输入：

```
import subprocess

print('$ nslookup')
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=su
output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
print(output.decode('utf-8'))
print('Exit code:', p.returncode)
```

上面的代码相当于在命令行执行命令 `nslookup` ，然后手动输入：

```
set q=mx
python.org
exit
```

运行结果如下：

```
$ nslookup
Server:         192.168.19.4
Address:     192.168.19.4#53

Non-authoritative answer:
python.org    mail exchanger = 50 mail.python.org.

Authoritative answers can be found from:
mail.python.org    internet address = 82.94.164.166
mail.python.org    has AAAA address 2001:888:2000:d::a6

Exit code: 0
```

进程间通信

`Process` 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的 `multiprocessing` 模块包装了底层的机制，提供了 `Queue` 、 `Pipes` 等多种方式来交换数据。

我们以 `Queue` 为例，在父进程中创建两个子进程，一个往 `Queue` 里写数据，一个从 `Queue` 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__=='__main__':
    # 父进程创建Queue，并传给各个子进程：
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

运行结果如下：

```
Process to write: 50563
Put A to queue...
Process to read: 50564
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

在Unix/Linux下，`multiprocessing`模块封装了`fork()`调用，使我们不需要关注`fork()`的细节。由于Windows没有`fork`调用，因此，`multiprocessing`需要“模拟”出`fork`的效果，父进程所有Python对象都必须通过pickle序列化再传到子进程去，所有，如果`multiprocessing`在Windows下调用失败了，要先考虑是不是pickle失败了。

小结

在Unix/Linux下，可以使用`fork()`调用实现多进程。

要实现跨平台的多进程，可以使用`multiprocessing`模块。

进程间通信是通过`Queue`、`Pipes`等实现的。

参考源码

do_folk.py

multi_processing.py

pooled_processing.py

do_subprocess.py

do_queue.py

多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

我们前面提到了进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：`_thread` 和 `threading`，`_thread` 是低级模块，`threading` 是高级模块，对 `_thread` 进行了封装。绝大多数情况下，我们只需要使用 `threading` 这个高级模块。

启动一个线程就是把一个函数传入并创建 `Thread` 实例，然后调用 `start()` 开始执行：

```
import time, threading

# 新线程执行的代码:
def loop():
    print('thread %s is running...' % threading.current_thread().na
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

执行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的 `threading` 模块有个 `current_thread()` 函数，它永远返回当前线程的实例。主线程实例的名字叫 `MainThread` ，子线程的名字在创建时指定，我们用 `LoopThread` 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为 `Thread-1` ， `Thread-2` ……

Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

```
import time, threading

# 假定这是你的银行存款:
balance = 0

def change_it(n):
    # 先存后取，结果应该为0:
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

我们定义了一个共享变量 `balance` ，初始值为 `0` ，并且启动两个线程，先存后取，理论上结果应该为 `0` ，但是，由于线程的调度是由操作系统决定的，当t1、t2交替执行时，只要循环次数足够多， `balance` 的结果就不一定是 `0` 了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

```
balance = balance + n
```

也分两步：

1. 计算 `balance + n` ，存入临时变量中；
2. 将临时变量的值赋给 `balance` 。

也就是可以看成：

```
x = balance + n
balance = x
```

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5
t1: balance = x1     # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1     # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2     # balance = 8
t2: x2 = balance - 8 # x2 = 8 - 8 = 0
t2: balance = x2     # balance = 0

结果 balance = 0
```

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

```
初始值 balance = 0

t1: x1 = balance + 5  # x1 = 0 + 5 = 5

t2: x2 = balance + 8  # x2 = 0 + 8 = 8
t2: balance = x2      # balance = 8

t1: balance = x1      # balance = 5
t1: x1 = balance - 5  # x1 = 5 - 5 = 0
t1: balance = x1      # balance = 0

t2: x2 = balance - 8  # x2 = 0 - 8 = -8
t2: balance = x2   # balance = -8

结果 balance = -8
```

究其原因，是因为修改 `balance` 需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改 `balance` 的时候，别的线程一定不能改。

如果我们要确保 `balance` 计算正确，就要给 `change_it()` 上一把锁，当某个线程开始执行 `change_it()` 时，我们说，该线程因为获得了锁，因此其他线程不能同时执行 `change_it()`，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过 `threading.Lock()` 来实现：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

当多个线程同时执行 `lock.acquire()` 时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 `try...finally` 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

多核CPU

如果你不幸拥有一个多核CPU，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

```
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

for i in range(multiprocessing.cpu_count()):
    t = threading.Thread(target=loop)
    t.start()
```

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有102%，也就是仅使用了一核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

小结

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

参考源码

multi_threading.py

do_lock.py

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
def process_student(name):
    std = Student(name)
    # std是局部变量，但是每个函数都要用它，因此必须传进去：
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的 `Student` 对象，不能共享。

如果用一个全局 `dict` 存放所有的 `Student` 对象，然后以 `thread` 自身作为 `key` 获得线程对应的 `Student` 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把std放到全局变量global_dict中：
    global_dict[threading.current_thread()] = std
    do_task_1()
    do_task_2()

def do_task_1():
    # 不传入std，而是根据当前线程查找：
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
    # 任何函数都可以查找出当前线程的std变量：
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 `std` 对象在每层函数中的传递问题，但是，每个函数获取 `std` 的代码有点丑。

有没有更简单的方式？

`ThreadLocal` 应运而生，不用查找 `dict` ， `ThreadLocal` 帮你自动做这件事：

```
import threading

# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().na

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name
t2 = threading.Thread(target= process_thread, args=('Bob',), name=
t1.start()
t2.start()
t1.join()
t2.join()
```

执行结果：

```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 `local_school` 就是一个 `ThreadLocal` 对象，每个 `Thread` 对它都可以读写 `student` 属性，但互不影响。你可以把 `local_school` 看成全局变量，但每个属性如 `local_school.student` 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题， `ThreadLocal` 内部会处理。

可以理解为全局变量 `local_school` 是一个 `dict` ，不但可以用 `local_school.student` ，还可以绑定其他变量，如 `local_school.teacher` 等等。

`ThreadLocal` 最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

小结

一个 `ThreadLocal` 变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。 `ThreadLocal` 解决了参数在一个线程中各个函数之间互相传递的问题。

参考源码

use_threadlocal.py

进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用 `fork` 调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示："该程序执行了非法操作，即将关闭"，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单进程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的 `multiprocessing` 模块不但支持多进程，其中 `managers` 子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于 `managers` 模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过 `Queue` 通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的 `Queue` 可以继续使用，但是，通过 `managers` 模块把 `Queue` 通过网络暴露出去，就可以让其他机器的进程访问 `Queue` 了。

我们先看服务进程，服务进程负责启动 `Queue` ，把 `Queue` 注册到网络上，然后往 `Queue` 里面写入任务：

```
# task_master.py

import random, time, queue
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = queue.Queue()
# 接收结果的队列:
result_queue = queue.Queue()

# 从BaseManager继承的QueueManager:
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:
QueueManager.register('get_task_queue', callable=lambda: task_queue
QueueManager.register('get_result_queue', callable=lambda: result_q
# 绑定端口5000, 设置验证码'abc':
manager = QueueManager(address=('', 5000), authkey=b'abc')
# 启动Queue:
manager.start()
# 获得通过网络访问的Queue对象:
task = manager.get_task_queue()
result = manager.get_result_queue()
# 放几个任务进去:
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)
# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
print('master exit.')
```

请注意，当我们在一台机器上写多进程程序时，创建的 `Queue` 可以直接拿来用，但是，在分布式多进程环境下，添加任务到 `Queue` 不可以直接对原始的 `task_queue` 进行操作，那样就绕过了 `QueueManager` 的封装，必须通过 `manager.get_task_queue()` 获得的 `Queue` 接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

```
# task_worker.py

import time, sys, queue
from multiprocessing.managers import BaseManager

# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器，也就是运行task_master.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与task_master.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务,并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动 `task_master.py` 服务进程：

```
$ python3 task_master.py
Put task 3411...
Put task 1605...
Put task 1398...
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...
Put task 7866...
Try get results...
```

`task_master.py` 进程发送完任务后，开始等待 `result` 队列的结果。现在启动 `task_worker.py` 进程：

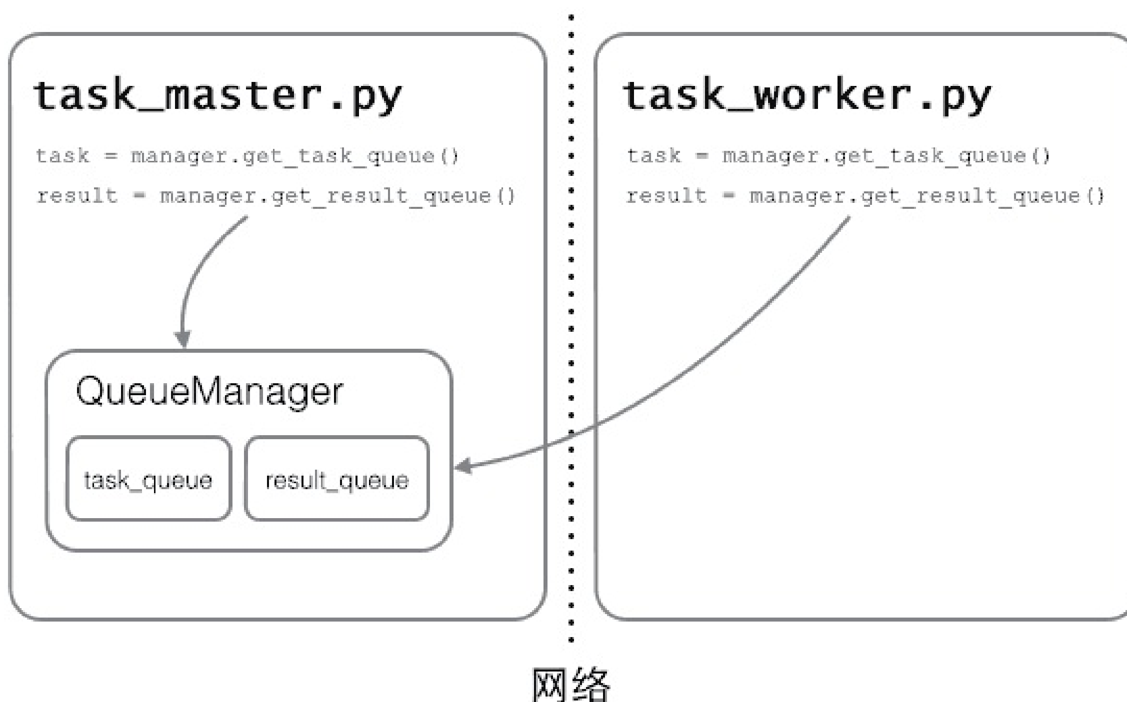
```
$ python3 task_worker.py
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

`task_worker.py` 进程结束，在 `task_master.py` 进程中会继续打印出结果：

```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算 `n*n` 的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到 `task_worker.py` 中根本没有创建Queue的代码，所以，Queue对象存储在 `task_master.py` 进程中：



而 `Queue` 之所以能通过网络访问，就是通过 `QueueManager` 实现的。由于 `QueueManager` 管理的不止一个 `Queue` ，所以，要给每个 `Queue` 的网络调用接口起个名字，比如 `get_task_queue` 。

`authkey` 有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果 `task_worker.py` 的 `authkey` 和 `task_master.py` 的 `authkey` 不一致，肯定连接不上。

小结

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。

注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

参考源码

task_master.py

task_worker.py

正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 `@` 前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 `\d` 可以匹配一个数字， `\w` 可以匹配一个字母或数字，所以：

- `'00\d'` 可以匹配 `'007'` ，但无法匹配 `'00A'` ；
- `'\d\d\d'` 可以匹配 `'010'` ；
- `'\w\w\d'` 可以匹配 `'py3'` ；

`.` 可以匹配任意字符，所以：

- `'py.'` 可以匹配 `'pyc'` 、 `'pyo'` 、 `'py!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括0个），用 `+` 表示至少一个字符，用 `?` 表示0个或1个字符，用 `{n}` 表示n个字符，用 `{n,m}` 表示n-m个字符：

来看一个复杂的例子： `\d{3}\s+\d{3,8}` 。

我们来从左到右解读一下：

1. `\d{3}` 表示匹配3个数字，例如 `'010'` ；

2. `\s` 可以匹配一个空格（也包括Tab等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '`， `'  '` 等；
3. `\d{3,8}` 表示3-8个数字，例如 `'1234567'` 。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `'\'` 转义，所以，上面的正则是 `\d{3}\-\d{3,8}` 。

但是，仍然无法匹配 `'010 - 12345'` ，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z\_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z\_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100'` ， `'0_Z'` ， `'Py3000'` 等等；
- `[a-zA-Z\_][0-9a-zA-Z\_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- `[a-zA-Z\_][0-9a-zA-Z\_]{0, 19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `[P|p]ython` 可以匹配 `'Python'` 或者 `'python'` 。

`^` 表示行的开头， `^\d` 表示必须以数字开头。

`$` 表示行的结束， `\d$` 表示必须以数字结束。

你可能注意到了， `py` 也可以匹配 `'python'` ，但是加上 `^py$` 就变成了整行匹配，就只能匹配 `'py'` 了。

re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供 `re` 模块，包含所有正则表达式的功能。由于Python的字符串本身也用 `\` 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\-001'
```

因此我们强烈建议使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None` 。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
>>> 'a b   c'.split(' ')
['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s+', 'a b   c')
['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入 `,` 试试：

```
>>> re.split(r'[\s\,]+', 'a,b, c  d')
['a', 'b', 'c', 'd']
```

再加入 `;` 试试：

```
>>> re.split(r'[\s\,\;]+', 'a,b;; c  d')
['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。

注意到 `group(0)` 永远是原始字符串， `group(1)` 、 `group(2)` ……表示第1、2、……个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2|
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'
```

对于 `'2-30'` ， `'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0` ：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。

必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用：
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

```
someone@gmail.com
bill.gates@microsoft.com
```

版本二可以验证并提取出带名字的Email地址：

```
<Tom Paris> tom@voyager.org
```

参考源码

regex.py

常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

本章将介绍一些常用的内建模块。

datetime

datetime是Python处理日期和时间的标准库。

获取当前日期和时间

我们先看如何获取当前日期和时间：

```
>>> from datetime import datetime
>>> now = datetime.now() # 获取当前datetime
>>> print(now)
2015-05-18 16:28:07.198690
>>> print(type(now))
<class 'datetime.datetime'>
```

注意到 `datetime` 是模块， `datetime` 模块还包含一个 `datetime` 类，通过 `from datetime import datetime` 导入的才是 `datetime` 这个类。

如果仅导入 `import datetime` ，则必须引用全名 `datetime.datetime` 。

`datetime.now()` 返回当前日期和时间，其类型是 `datetime` 。

获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个 `datetime` ：

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> print(dt)
2015-04-19 12:20:00
```

datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把1970年1月1日 00:00:00 UTC+00:00时区的时刻称为epoch time，记为 `0` （1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

你可以认为：

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

可见timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

把一个 `datetime` 类型转换为timestamp只需要简单调用 `timestamp()` 方法：

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> dt.timestamp() # 把timestamp转换为datetime
1429417200.0
```

注意Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

timestamp转换为datetime

要把timestamp转换为 `datetime` ，使用 `datetime` 提供的 `fromtimestamp()` 方法：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t))
2015-04-19 12:20:00
```

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是UTC+8:00时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp也可以直接被转换到UTC标准时区的时间：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t)) # 本地时间
2015-04-19 12:20:00
>>> print(datetime.utcfromtimestamp(t)) # UTC时间
2015-04-19 04:20:00
```

str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过 `datetime.strptime()` 实现，需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%
>>> print(cday)
2015-06-01 18:19:59
```

字符串 `'%Y-%m-%d %H:%M:%S'` 规定了日期和时间部分的格式。详细的说明请参考Python文档。

注意转换后的datetime是没有时区信息的。

datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过 `strftime()` 实现的，同样需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> print(now.strftime('%a, %b %d %H:%M'))
Mon, May 05 16:28
```

datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用 `+` 和 `-` 运算符，不过需要导入 `timedelta` 这个类：

```
>>> from datetime import datetime, timedelta
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
>>> now + timedelta(hours=10)
datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
>>> now - timedelta(days=1)
datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
>>> now + timedelta(days=2, hours=12)
datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

可见，使用 `timedelta` 你可以很容易地算出前几天和后几天的时刻。

本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个 `datetime` 类型有一个时区属性 `tzinfo` ，但是默认为 `None` ，所以无法区分这个 `datetime` 到底是哪个时区，除非强行给 `datetime` 设置一个时区：

```
>>> from datetime import datetime, timedelta, timezone
>>> tz_utc_8 = timezone(timedelta(hours=8)) # 创建时区UTC+8:00
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)
>>> dt = now.replace(tzinfo=tz_utc_8) # 强制设置为UTC+8:00
>>> dt
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.t
```

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

时区转换

我们可以先通过 `utcnow()` 拿到当前的UTC时间，再转换为任意时区的时间：

```
# 拿到UTC时间，并强制设置时区为UTC+0:00:
>>> utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
>>> print(utc_dt)
2015-05-18 09:05:12.377316+00:00
# astimezone()将转换时区为北京时间:
>>> bj_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
>>> print(bj_dt)
2015-05-18 17:05:12.377316+08:00
# astimezone()将转换时区为东京时间:
>>> tokyo_dt = utc_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt)
2015-05-18 18:05:12.377316+09:00
# astimezone()将bj_dt转换时区为东京时间:
>>> tokyo_dt2 = bj_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt2)
2015-05-18 18:05:12.377316+09:00
```

时区转换的关键在于，拿到一个 `datetime` 时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的 `datetime` ，通过 `astimezone()` 方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的 `datetime` 都可以正确转换，例如上述 `bj_dt` 到 `tokyo_dt` 的转换。

小结

`datetime` 表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。

如果要存储 `datetime` ，最佳方法是将其转换为timestamp再存储，因为timestamp的值与时区完全无关。

练习

假设你获取了用户输入的日期和时间如 `2015-1-21 9:01:30` ，以及一个时区信息如 `UTC+5:00` ，均是 `str` ，请编写一个函数将其转换为timestamp：

```
# -*- coding:utf-8 -*-

import re
from datetime import datetime, timezone, timedelta

def to_timestamp(dt_str, tz_str):

    pass

# 测试:

t1 = to_timestamp('2015-6-1 08:10:30', 'UTC+7:00')
assert t1 == 1433121030.0, t1

t2 = to_timestamp('2015-5-31 16:10:30', 'UTC-09:00')
assert t2 == 1433121030.0, t2

print('Pass')
```

参考源码

use_datetime.py

collections

collections是Python内建的一个集合模块，提供了许多有用的集合类。

namedtuple

我们知道 `tuple` 可以表示不变集合，例如，一个点的二维坐标就可以表示成：

```
>>> p = (1, 2)
```

但是，看到 `(1, 2)` ，很难看出这个 `tuple` 是用来表示一个坐标的。

定义一个class又小题大做了，这时， `namedtuple` 就派上了用场：

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。

这样一来，我们用 `namedtuple` 可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的 `Point` 对象是 `tuple` 的一种子类：

```
>>> isinstance(p, Point)
True
>>> isinstance(p, tuple)
True
```

类似的，如果要用坐标和半径表示一个圆，也可以用 `namedtuple` 定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

deque

使用 `list` 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 `list` 是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

`deque` 除了实现list的 `append()` 和 `pop()` 外，还支持 `appendleft()` 和 `popleft()` ，这样就可以非常高效地往头部添加或删除元素。

defaultdict

使用 `dict` 时，如果引用的Key不存在，就会抛出 `KeyError` 。如果希望key不存在时，返回一个默认值，就可以用 `defaultdict` ：

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1存在
'abc'
>>> dd['key2'] # key2不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建 `defaultdict` 对象时传入。

除了在Key不存在时返回默认值， `defaultdict` 的其他行为跟 `dict` 是完全一样的。

OrderedDict

使用 `dict` 时，Key是无序的。在对 `dict` 做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用 `OrderedDict` ：

```
>>> from collections import OrderedDict
>>> d = dict([('a', 1), ('b', 2), ('c', 3)])
>>> d # dict的Key是无序的
{'a': 1, 'c': 3, 'b': 2}
>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> od # OrderedDict的Key是有序的
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

注意， `OrderedDict` 的Key会按照插入的顺序排列，不是Key本身排序：

```
>>> od = OrderedDict()
>>> od['z'] = 1
>>> od['y'] = 2
>>> od['x'] = 3
>>> list(od.keys()) # 按照插入的Key的顺序返回
['z', 'y', 'x']
```

`OrderedDict` 可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

```
from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):

    def __init__(self, capacity):
        super(LastUpdatedOrderedDict, self).__init__()
        self._capacity = capacity

    def __setitem__(self, key, value):
        containsKey = 1 if key in self else 0
        if len(self) - containsKey >= self._capacity:
            last = self.popitem(last=False)
            print('remove:', last)
        if containsKey:
            del self[key]
            print('set:', (key, value))
        else:
            print('add:', (key, value))
        OrderedDict.__setitem__(self, key, value)
```

Counter

`Counter` 是一个简单的计数器，例如，统计字符出现的个数：

```
>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p
```

`Counter` 实际上也是 `dict` 的一个子类，上面的结果可以看出，字符 `'g'` 、 `'m'` 、 `'r'` 各出现了两次，其他字符各出现了一次。

小结

`collections`模块提供了一些有用的集合类，可以根据需要选用。

参考源码

[use_collections.py]

base64

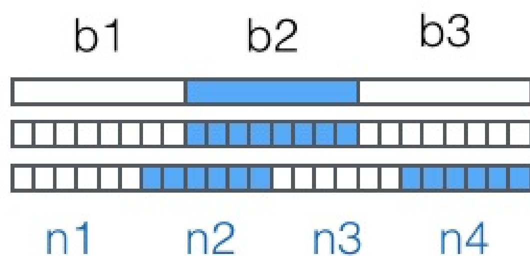
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开 `exe` 、 `jpg` 、 `pdf` 这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

```
['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']
```

然后，对二进制数据进行处理，每3个字节一组，一共是 `3x8=24` bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用 `\x00` 字节在末尾补足后，再在编码的末尾加上1个或2个 `=` 号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的 `base64` 可以直接进行base64的编解码：

```
>>> import base64
>>> base64.b64encode(b'binary\x00string')
b'YmluYXJ5AHN0cmluZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')
b'binary\x00string'
```

由于标准的Base64编码后可能出现字符 `+` 和 `/` ，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符 `+` 和 `/` 分别变成 `-` 和 `_` ：

```
>>> base64.b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd++//'
>>> base64.urlsafe_b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd--__'
>>> base64.urlsafe_b64decode('abcd--__')
b'i\xb7\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于 `=` 字符也可能出现在Base64编码中，但 `=` 用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把 `=` 去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉 `=` 后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上 `=` 把Base64字符串的长度变为4的倍数，就可以正常解码了。

小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

练习

请写一个能处理去掉`=`的base64解码函数：

```
# -*- coding: utf-8 -*-

import base64

def safe_base64_decode(s):

    pass

# 测试:
assert b'abcd' == safe_base64_decode(b'YWJjZA=='), safe_base64_decc
assert b'abcd' == safe_base64_decode(b'YWJjZA'), safe_base64_decode
print('Pass')
```

参考源码

do_base64.py

struct

准确地讲，Python没有专门处理字节的数据类型。但由于 `str` 既是字符串，又可以表示字节，所以，字节数组＝str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的 `bytes` ，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = (n & 0xff000000) >> 24
>>> b2 = (n & 0xff0000) >> 16
>>> b3 = (n & 0xff00) >> 8
>>> b4 = n & 0xff
>>> bs = bytes([b1, b2, b3, b4])
>>> bs
b'\x00\x9c@c'
```

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个 `struct` 模块来解决 `bytes` 和其他二进制数据类型的转换。

`struct` 的 `pack` 函数把任意数据类型变成 `bytes` ：

```
>>> import struct
>>> struct.pack('>I', 10240099)
b'\x00\x9c@c'
```

`pack` 的第一个参数是处理指令， `'>I'` 的意思是：

`>` 表示字节顺序是big-endian，也就是网络序， `I` 表示4字节无符号整数。

后面的参数个数要和处理指令一致。

`unpack` 把 `bytes` 变成相应的数据类型：

```
>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')
(4042322160, 32896)
```

根据 `>IH` 的说明，后面的 `bytes` 依次变为 `I` ：4字节无符号整数和 `H` ：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用 `struct` 就方便多了。

`struct` 模块定义的数据类型可以参考Python官方文档：

https://docs.python.org/3/library/struct.html#format-characters

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用 `struct` 分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

```
>>> s = b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\
```

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节： `'BM'` 表示Windows位图， `'BA'` 表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用 `unpack` 读取：

```
>>> struct.unpack('<ccIIIIIIHH', s)
(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示， `b'B'` 、 `b'M'` 说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个 `bmpinfo.py` ，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

参考源码

check_bmp.py

hashlib

摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串`'how to use python hashlib - by Michael'`，并附上这篇文章的摘要是`'2d73d4f15c0db7f5ecb321b6a65e5d6d'`。如果有人篡改了你的文章，并发表为`'how to use python hashlib - by Bob'`，你可以一下子指出Bob篡改了你的文章，因为根据`'how to use python hashlib - by Bob'`计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数`f()`对任意长度的数据`data`计算出固定长度的摘要`digest`，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算`f(data)`很容易，但通过`digest`反推`data`却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用`update()`，最后计算的结果是一样的：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in '.encode('utf-8'))
md5.update('python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in '.encode('utf-8'))
sha1.update('python hashlib?'.encode('utf-8'))
print(sha1.hexdigest())
```

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章 `'how to learn hashlib in python - by Bob'`，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

```
name    | password
--------+----------
michael | 123456
bob     | abc999
alice   | alice2008
```

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

```
username | password
---------+---------------------------------
michael  | e10adc3949ba59abbe56e057f20f883e
bob      | 878ef96e86145580c38c87f0410ad153
alice    | 99b1c2188db85afee403b1536010c2c9
```

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

练习

根据用户输入的口令，计算出存储在数据库中的MD5口令：

```
def calc_md5(password):
    pass
```

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

```
db = {
    'michael': 'e10adc3949ba59abbe56e057f20f883e',
    'bob': '878ef96e86145580c38c87f0410ad153',
    'alice': '99b1c2188db85afee403b1536010c2c9'
}

def login(user, password):
    pass
```

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用 `123456` ， `888888` ， `password` 这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

```
'e10adc3949ba59abbe56e057f20f883e': '123456'
'21218cca77804d2ba1922c33e0151105': '888888'
'5f4dcc3b5aa765d61d8327deb882cf99': 'password'
```

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

```
def calc_md5(password):
    return get_md5(password + 'the-Salt')
```

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如`123456`，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

练习

根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

```
db = {}

def register(username, password):
    db[username] = get_md5(password + username + 'the-Salt')
```

然后，根据修改后的MD5算法实现用户登录的验证：

```
def login(username, password):
    pass
```

小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

参考源码

use_hashlib.py

itertools

Python的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。

首先，我们看看 `itertools` 提供的几个“无限”迭代器：

```
>>> import itertools
>>> natuals = itertools.count(1)
>>> for n in natuals:
...     print(n)
...
1
2
3
...
```

因为 `count()` 会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按 `Ctrl+C` 退出。

`cycle()` 会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print(c)
...
'A'
'B'
'C'
'A'
'B'
'C'
...
```

同样停不下来。

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
>>> ns = itertools.repeat('A', 3)
>>> for n in ns:
...     print(n)
...
A
A
A
```

无限序列只有在 `for` 迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过 `takewhile()` 等函数根据条件判断来截取出一个有限的序列：

```
>>> natuals = itertools.count(1)
>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)
>>> list(ns)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`itertools` 提供的几个迭代器操作函数更加有用：

chain()

`chain()` 可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
>>> for c in itertools.chain('ABC', 'XYZ'):
...     print(c)
# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'
```

groupby()

`groupby()` 把迭代器中相邻的重复元素挑出来放在一起：

```
>>> for key, group in itertools.groupby('AAABBBCCAAA'):
...     print(key, list(group))
...
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素 `'A'` 和 `'a'` 都返回相同的key：

```
>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.
...     print(key, list(group))
...
A ['A', 'a', 'a']
B ['B', 'B', 'b']
C ['c', 'C']
A ['A', 'A', 'a']
```

小结

`itertools` 模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是 `Iterator` ，只有用 `for` 循环迭代的时候才真正计算。

参考源码

use_itertools.py

XML

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是`start_element`，`end_element`和`char_data`，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

```
<a href="/">python</a>
```

会产生3个事件：

1. start_element事件，在读取`<a href="/">`时；
2. char_data事件，在读取`python`时；
3. end_element事件，在读取`</a>`时。

用代码实验一下：

```
from xml.parsers.expat import ParserCreate

class DefaultSaxHandler(object):
    def start_element(self, name, attrs):
        print('sax:start_element: %s, attrs: %s' % (name, str(attrs

    def end_element(self, name):
        print('sax:end_element: %s' % name)

    def char_data(self, text):
        print('sax:char_data: %s' % text)

xml = r'''<?xml version="1.0"?>
<ol>
    <li><a href="/python">Python</a></li>
    <li><a href="/ruby">Ruby</a></li>
</ol>
'''

handler = DefaultSaxHandler()
parser = ParserCreate()
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)
```

需要注意的是读取一大段字符串时， `CharacterDataHandler` 可能被多次调用，所以需要自己保存起来，在 `EndElementHandler` 里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

```
L = []
L.append(r'<?xml version="1.0"?>')
L.append(r'<root>')
L.append(encode('some & data'))
L.append(r'</root>')
return ''.join(L)
```

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

练习

请利用SAX编写程序解析Yahoo的XML格式的天气预报，获取当天和第二天的天气：

http://weather.yahooapis.com/forecastrss?u=c&w=2151330

参数 `w` 是城市代码，要查询某个城市代码，可以在weather.yahoo.com搜索城市，浏览器地址栏的URL就包含城市代码。

```
# -*- coding:utf-8 -*-

from xml.parsers.expat import ParserCreate

class WeatherSaxHandler(object):
    pass

def parse_weather(xml):
    return {
        'city': 'Beijing',
        'country': 'China',
        'today': {
            'text': 'Partly Cloudy',
            'low': 20,
            'high': 33
        },
        'tomorrow': {
            'text': 'Sunny',
            'low': 21,
            'high': 34
        }
```

```
    }

# 测试:
data = r'''<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/
    <channel>
        <title>Yahoo! Weather - Beijing, CN</title>
        <lastBuildDate>Wed, 27 May 2015 11:00 am CST</lastBuildDate
        <yweather:location city="Beijing" region="" country="China"
        <yweather:units temperature="C" distance="km" pressure="mb"
        <yweather:wind chill="28" direction="180" speed="14.48" />
        <yweather:atmosphere humidity="53" visibility="2.61" pressu
        <yweather:astronomy sunrise="4:51 am" sunset="7:32 pm"/>
        <item>
            <geo:lat>39.91</geo:lat>
            <geo:long>116.39</geo:long>
            <pubDate>Wed, 27 May 2015 11:00 am CST</pubDate>
            <yweather:condition text="Haze" code="21" temp="28" dat
            <yweather:forecast day="Wed" date="27 May 2015" low="20
            <yweather:forecast day="Thu" date="28 May 2015" low="21
            <yweather:forecast day="Fri" date="29 May 2015" low="18
            <yweather:forecast day="Sat" date="30 May 2015" low="18
            <yweather:forecast day="Sun" date="31 May 2015" low="20
        </item>
    </channel>
</rss>
'''
weather = parse_weather(data)
assert weather['city'] == 'Beijing', weather['city']
assert weather['country'] == 'China', weather['country']
assert weather['today']['text'] == 'Partly Cloudy', weather['today'
assert weather['today']['low'] == 20, weather['today']['low']
assert weather['today']['high'] == 33, weather['today']['high']
assert weather['tomorrow']['text'] == 'Sunny', weather['tomorrow'][
assert weather['tomorrow']['low'] == 21, weather['tomorrow']['low']
assert weather['tomorrow']['high'] == 34, weather['tomorrow']['high
print('Weather:', str(weather))
```

参考源码

use_sax.py

HTMLParser

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print('<%s>' % tag)

    def handle_endtag(self, tag):
        print('</%s>' % tag)

    def handle_startendtag(self, tag, attrs):
        print('<%s/>' % tag)

    def handle_data(self, data):
        print(data)

    def handle_comment(self, data):
        print('<!--', data, '-->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
        print('&#%s;' % name)

parser = MyHTMLParser()
parser.feed('''<html>
<head></head>
<body>
<!-- test html parser -->
    <p>Some <a href=\"#\">html</a> HTML tutorial...<br>END</p>
</body></html>''')
```

`feed()` 方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的` `，一种是数字表示的`Ӓ`，这两种字符都可以通过Parser解析出来。

小结

利用HTMLParser，可以把网页中的文本、图像等解析出来。

练习

找一个网页，例如https://www.python.org/events/python-events/，用浏览器查看源码并复制，然后尝试解析一下HTML，输出Python官网发布的会议时间、名称和地点。

参考源码

use_htmlparser.py

urllib

urllib提供了一系列用于操作URL的功能。

Get

urllib的 `request` 模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URL `https://api.douban.com/v2/book/2129650` 进行抓取，并返回响应：

```
from urllib import request

with request.urlopen('https://api.douban.com/v2/book/2129650') as f
    data = f.read()
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', data.decode('utf-8'))
```

可以看到HTTP响应的头和JSON数据：

```
Status: 200 OK
Server: nginx
Date: Tue, 26 May 2015 10:02:27 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2049
Connection: close
Expires: Sun, 1 Jan 2006 01:00:00 GMT
Pragma: no-cache
Cache-Control: must-revalidate, no-cache, private
X-DAE-Node: pidl1
Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},'
```

如果我们要想模拟浏览器发送GET请求，就需要使用 `Request` 对象，通过往 `Request` 对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

```
from urllib import request

req = request.Request('http://www.douban.com/')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_
with request.urlopen(req) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

这样豆瓣会返回适合iPhone的移动版网页：

```
...
    <meta name="viewport" content="width=device-width, user-scalabl
    <meta name="format-detection" content="telephone=no">
    <link rel="apple-touch-icon" sizes="57x57" href="http://img4.dc
...
```

Post

如果要以POST发送一个请求，只需要把参数 `data` 以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以 `username=xxx&password=xxx` 的编码传入：

```
from urllib import request, parse

print('Login to weibo.cn...')
email = input('Email: ')
passwd = input('Password: ')
login_data = parse.urlencode([
    ('username', email),
    ('password', passwd),
    ('entry', 'mweibo'),
    ('client_id', ''),
    ('savestate', '1'),
    ('ec', ''),
    ('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=r
])

req = request.Request('https://passport.weibo.cn/sso/login')
req.add_header('Origin', 'https://passport.weibo.cn')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_
req.add_header('Referer', 'https://passport.weibo.cn/signin/login?e

with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

如果登录成功，我们获得的响应如下：

```
Status: 200 OK
Server: nginx/1.2.0
...
Set-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn
...
Data: {"retcode":20000000,"msg":"","data":{...,"uid":"1658384301"}}
```

如果登录失败，我们获得的响应如下：

```
...
Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u78
```

Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用 `ProxyHandler` 来处理，示例代码如下：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.ex
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'passw
opener = urllib.request.build_opener(proxy_handler, proxy_auth_hanc
with opener.open('http://www.example.com/login.html') as f:
    pass
```

小结

urllib提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪装， `User-Agent` 头就是用来标识浏览器的。

练习

利用urllib读取XML，将XML一节的数据由硬编码改为由urllib获取：

```
from urllib import request, parse

def fetch_xml(url):

    pass

# 测试
print(fetch_xml('http://weather.yahooapis.com/forecastrss?u=c&w=215
```

参考源码

use_urllib.py

常用第三方模块

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在PyPI - the Python Package Index上注册，只要找到对应的模块名字，即可用pip安装。

本章介绍常用的第三方模块。

PIL

PIL：Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫Pillow，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow。

安装Pillow

在命令行下直接通过pip安装：

```
$ pip install pillow
```

如果遇到 `Permission denied` 安装失败，请加上 `sudo` 重试。

操作图像

来看看最常见的图像缩放操作，只需三四行代码：

```
from PIL import Image

# 打开一个jpg图像文件，注意是当前路径:
im = Image.open('test.jpg')
# 获得图像尺寸:
w, h = im.size
print('Original image size: %sx%s' % (w, h))
# 缩放到50%:
im.thumbnail((w//2, h//2))
print('Resize image to: %sx%s' % (w//2, h//2))
# 把缩放后的图像用jpeg格式保存:
im.save('thumbnail.jpg', 'jpeg')
```

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

```
from PIL import Image, ImageFilter

# 打开一个jpg图像文件，注意是当前路径:
im = Image.open('test.jpg')
# 应用模糊滤镜:
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

效果如下：



PIL的 `ImageDraw` 提供了一系列绘图方法，让我们可以直接绘图。比如要生成字母验证码图片：

```
from PIL import Image, ImageDraw, ImageFont, ImageFilter

import random

# 随机字母:
def rndChar():
    return chr(random.randint(65, 90))

# 随机颜色1:
def rndColor():
    return (random.randint(64, 255), random.randint(64, 255), rando

# 随机颜色2:
def rndColor2():
    return (random.randint(32, 127), random.randint(32, 127), rando

# 240 x 60:
width = 60 * 4
height = 60
image = Image.new('RGB', (width, height), (255, 255, 255))
# 创建Font对象:
font = ImageFont.truetype('Arial.ttf', 36)
# 创建Draw对象:
draw = ImageDraw.Draw(image)
# 填充每个像素:
for x in range(width):
    for y in range(height):
        draw.point((x, y), fill=rndColor())
# 输出文字:
for t in range(4):
    draw.text((60 * t + 10, 10), rndChar(), font=font, fill=rndColo
# 模糊:
image = image.filter(ImageFilter.BLUR)
image.save('code.jpg', 'jpeg')
```

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：



如果运行的时候报错：

```
IOError: cannot open resource
```

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

```
'/Library/Fonts/Arial.ttf'
```

要详细了解PIL的强大功能，请请参考Pillow官方文档：

https://pillow.readthedocs.org/

小结

PIL提供了操作图像的强大功能，可以通过简单的代码完成复杂的图像处理。

参考源码

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_resize.py

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_blur.py

https://github.com/michaelliao/learn-python3/blob/master/samples/packages/pil/use_pil_draw.py

virtualenv

在开发Python应用程序的时候，系统安装的Python3只有一个版本：3.4。所有第三方的包都会被 `pip` 安装到Python3的 `site-packages` 目录下。

如果我们要同时开发多个应用程序，那这些应用程序都会共用一个Python，就是安装在系统的Python 3。如果应用A需要jinja 2.7，而应用B需要jinja 2.6怎么办？

这种情况下，每个应用可能需要各自拥有一套“独立”的Python运行环境。virtualenv就是用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用 `pip` 安装virtualenv：

```
$ pip3 install virtualenv
```

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

```
Mac:~ michael$ mkdir myproject
Mac:~ michael$ cd myproject/
Mac:myproject michael$
```

第二步，创建一个独立的Python运行环境，命名为 `venv` ：

```
Mac:myproject michael$ virtualenv --no-site-packages venv
Using base prefix '/usr/local/.../Python.framework/Versions/3.4'
New python executable in venv/bin/python3.4
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

命令 `virtualenv` 就可以创建一个独立的Python运行环境，我们还加上了参数 `--no-site-packages` ，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的`venv`目录。有了`venv`这个Python环境，可以用`source`进入该环境：

```
Mac:myproject michael$ source venv/bin/activate
(venv)Mac:myproject michael$
```

注意到命令提示符变了，有个`(venv)`前缀，表示当前环境是一个名为`venv`的Python环境。

下面正常安装各种第三方包，并运行`python`命令：

```
(venv)Mac:myproject michael$ pip install jinja2
...
Successfully installed jinja2-2.7.3 markupsafe-0.23
(venv)Mac:myproject michael$ python myapp.py
...
```

在`venv`环境下，用`pip`安装的包都被安装到`venv`这个环境下，系统Python环境不受任何影响。也就是说，`venv`环境是专门针对`myproject`这个应用创建的。

退出当前的`venv`环境，使用`deactivate`命令：

```
(venv)Mac:myproject michael$ deactivate
Mac:myproject michael$
```

此时就回到了正常的环境，现在`pip`或`python`均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令`source venv/bin/activate`进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令`python`和`pip`均指向当前的virtualenv环境。

小结

virtualenv为应用提供了隔离的Python运行环境，解决了不同应用间多版本的冲突问题。

图形界面

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets
- Qt
- GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

第一步是导入Tkinter包的所有内容：

```
from tkinter import *
```

第二步是从 `Frame` 派生一个 `Application` 类，这是所有Widget的父容器：

```
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.helloLabel = Label(self, text='Hello, world!')
        self.helloLabel.pack()
        self.quitButton = Button(self, text='Quit', command=self.qu
        self.quitButton.pack()
```

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

`pack()` 方法把Widget加入到父容器中，并实现布局。 `pack()` 是最简单的布局， `grid()` 可以实现更复杂的布局。

在 `createWidgets()` 方法中，我们创建一个 `Label` 和一个 `Button` ，当Button被点击时，触发 `self.quit()` 使程序退出。

第三步，实例化 `Application` ，并启动消息循环：

```
app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

运行这个GUI程序，可以看到下面的窗口：



点击“Quit”按钮或者窗口的“x”结束程序。

输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

```
from tkinter import *
import tkinter.messagebox as messagebox

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

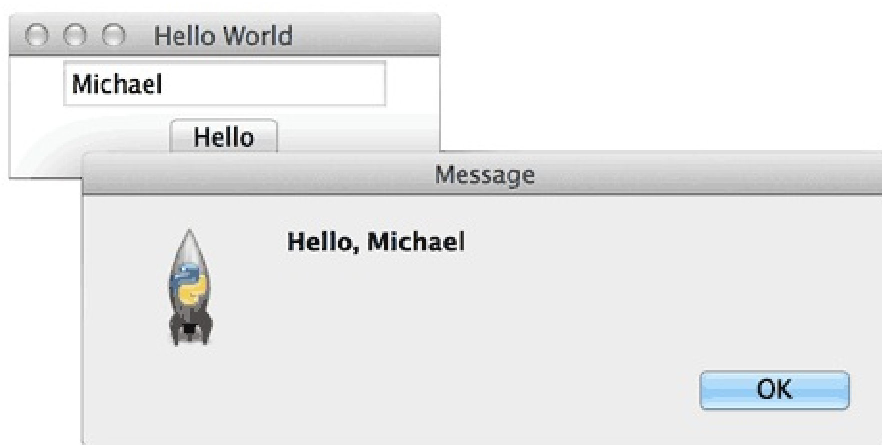
    def createWidgets(self):
        self.nameInput = Entry(self)
        self.nameInput.pack()
        self.alertButton = Button(self, text='Hello', command=self.
        self.alertButton.pack()

    def hello(self):
        name = self.nameInput.get() or 'world'
        messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()
# 设置窗口标题:
app.master.title('Hello World')
# 主消息循环:
app.mainloop()
```

当用户点击按钮时，触发 `hello()` ，通过 `self.nameInput.get()` 获得用户输入的文本后，使用 `tkMessageBox.showinfo()` 可以弹出消息对话框。

程序运行结果如下：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

参考源码

hello_gui.py

网络编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。



网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

本章我们将详细介绍Python网络编程的概念和最主要的两种网络类型的编程。

TCP/IP简介

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

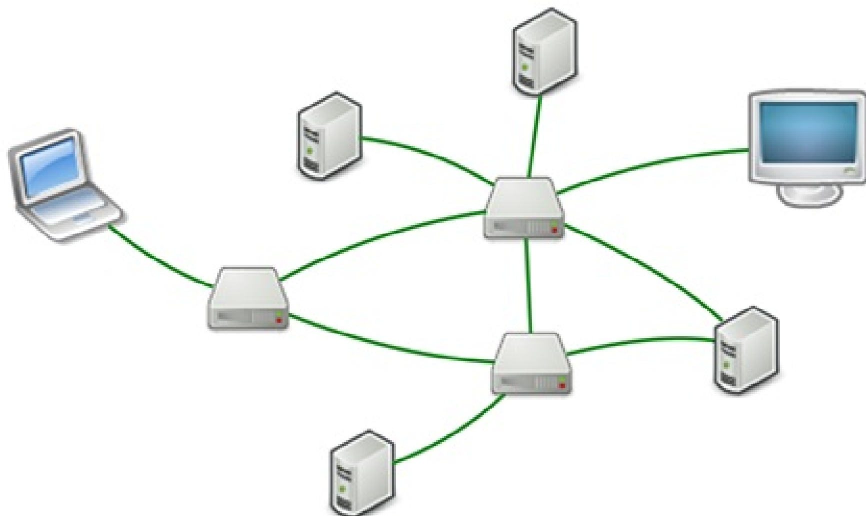
计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似 `123.123.123.123` 。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。



IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如 `192.168.0.1` 实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于 `2001:0db8:85a3:0042:1000:8a2e:0370:7334` 。

TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个IP包除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个网络程序。一个IP包来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。

TCP编程

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

客户端

大多数连接都是可靠的TCP连接。创建TCP连接时，主动发起连接的叫客户端，被动响应连接的叫服务器。

举个例子，当我们在浏览器中访问新浪时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

所以，我们要创建一个基于TCP连接的Socket，可以这样做：

```
# 导入socket库:
import socket

# 创建一个socket:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('www.sina.com.cn', 80))
```

创建 `Socket` 时， `AF_INET` 指定使用IPv4协议，如果要用更先进的IPv6，就指定为 `AF_INET6` 。 `SOCK_STREAM` 指定使用面向流的TCP协议，这样，一个 `Socket` 对象就创建成功，但是还没有建立连接。

客户端要主动发起TCP连接，必须知道服务器的IP地址和端口号。新浪网站的IP地址可以用域名 `www.sina.com.cn` 自动转换到IP地址，但是怎么知道新浪服务器的端口号呢？

答案是作为服务器，提供什么样的服务，端口号就必须固定下来。由于我们想要访问网页，因此新浪提供网页服务的服务器必须把端口号固定在 `80` 端口，因为 `80` 端口是Web服务的标准端口。其他服务都有对应的标准端口号，例如SMTP

服务是 `25` 端口，FTP服务是 `21` 端口，等等。端口号小于1024的是Internet标准服务的端口，端口号大于1024的，可以任意使用。

因此，我们连接新浪服务器的代码如下：

```
s.connect(('www.sina.com.cn', 80))
```

注意参数是一个 `tuple` ，包含地址和端口号。

建立TCP连接后，我们就可以向新浪服务器发送请求，要求返回首页的内容：

```
# 发送数据:
s.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: clo
```

TCP连接创建的是双向通道，双方都可以同时给对方发数据。但是谁先发谁后发，怎么协调，要根据具体的协议来决定。例如，HTTP协议规定客户端必须先发请求给服务器，服务器收到后才发数据给客户端。

发送的文本格式必须符合HTTP标准，如果格式没问题，接下来就可以接收新浪服务器返回的数据了：

```
# 接收数据:
buffer = []
while True:
    # 每次最多接收1k字节:
    d = s.recv(1024)
    if d:
        buffer.append(d)
    else:
        break
data = b''.join(buffer)
```

接收数据时，调用 `recv(max)` 方法，一次最多接收指定的字节数，因此，在一个while循环中反复接收，直到 `recv()` 返回空数据，表示接收完毕，退出循环。

当我们接收完数据后，调用 `close()` 方法关闭Socket，这样，一次完整的网络通信就结束了：

```
# 关闭连接:
s.close()
```

接收到的数据包括HTTP头和网页本身，我们只需要把HTTP头和网页分离一下，把HTTP头打印出来，网页内容保存到文件：

```
header, html = data.split(b'\r\n\r\n', 1)
print(header.decode('utf-8'))
# 把接收的数据写入文件:
with open('sina.html', 'wb') as f:
    f.write(html)
```

现在，只需要在浏览器中打开这个 `sina.html` 文件，就可以看到新浪的首页了。

服务器

和客户端编程相比，服务器编程就要复杂一些。

服务器进程首先要绑定一个端口并监听来自其他客户端的连接。如果某个客户端连接过来了，服务器就与该客户端建立Socket连接，随后的通信就靠这个Socket连接了。

所以，服务器会打开固定端口（比如80）监听，每来一个客户端连接，就创建该Socket连接。由于服务器会有大量来自客户端的连接，所以，服务器要能够区分一个Socket连接是和哪个客户端绑定的。一个Socket依赖4项：服务器地址、服务器端口、客户端地址、客户端端口来唯一确定一个Socket。

但是服务器还需要同时响应多个客户端的请求，所以，每个连接都需要一个新的进程或者新的线程来处理，否则，服务器一次就只能服务一个客户端了。

我们来编写一个简单的服务器程序，它接收客户端连接，把客户端发过来的字符串加上 `Hello` 再发回去。

首先，创建一个基于IPv4和TCP协议的Socket：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

然后，我们要绑定监听的地址和端口。服务器可能有多块网卡，可以绑定到某一块网卡的IP地址上，也可以用 `0.0.0.0` 绑定到所有的网络地址，还可以用 `127.0.0.1` 绑定到本机地址。 `127.0.0.1` 是一个特殊的IP地址，表示本机地址，如果绑定到这个地址，客户端必须同时在本机运行才能连接，也就是说，外部的计算机无法连接进来。

端口号需要预先指定。因为我们写的这个服务不是标准服务，所以用 `9999` 这个端口号。请注意，小于 `1024` 的端口号必须要有管理员权限才能绑定：

```
# 监听端口:
s.bind(('127.0.0.1', 9999))
```

紧接着，调用 `listen()` 方法开始监听端口，传入的参数指定等待连接的最大数量：

```
s.listen(5)
print('Waiting for connection...')
```

接下来，服务器程序通过一个永久循环来接受来自客户端的连接， `accept()` 会等待并返回一个客户端的连接:

```
while True:
    # 接受一个新连接:
    sock, addr = s.accept()
    # 创建新线程来处理TCP连接:
    t = threading.Thread(target=tcplink, args=(sock, addr))
    t.start()
```

每个连接都必须创建新线程（或进程）来处理，否则，单线程在处理连接的过程中，无法接受其他客户端的连接：

```
def tcplink(sock, addr):
    print('Accept new connection from %s:%s...' % addr)
    sock.send(b'Welcome!')
    while True:
        data = sock.recv(1024)
        time.sleep(1)
        if not data or data.decode('utf-8') == 'exit':
            break
        sock.send(('Hello, %s!' % data.decode('utf-8')).encode('utf
    sock.close()
    print('Connection from %s:%s closed.' % addr)
```

连接建立后，服务器首先发一条欢迎消息，然后等待客户端数据，并加上 `Hello` 再发送给客户端。如果客户端发送了 `exit` 字符串，就直接关闭连接。

要测试这个服务器程序，我们还需要编写一个客户端程序：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 建立连接:
s.connect(('127.0.0.1', 9999))
# 接收欢迎消息:
print(s.recv(1024).decode('utf-8'))
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据:
    s.send(data)
    print(s.recv(1024).decode('utf-8'))
s.send(b'exit')
s.close()
```

我们需要打开两个命令行窗口，一个运行服务器程序，另一个运行客户端程序，就可以看到效果了：



需要注意的是，客户端程序运行完毕就退出了，而服务器程序会永远运行下去，必须按Ctrl+C退出程序。

小结

用TCP协议进行Socket编程在Python中十分简单，对于客户端，要主动连接服务器的IP和指定端口，对于服务器，要首先监听指定端口，然后，对每一个新的连接，创建一个线程或进程来处理。通常，服务器程序会无限运行下去。

同一个端口，被一个Socket绑定了以后，就不能被别的Socket绑定了。

参考源码

do_tcp.py

UDP编程

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

我们来看看如何通过UDP协议传输数据。和TCP类似，使用UDP的通信双方也分为客户端和服务器。服务器首先需要绑定端口：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# 绑定端口:
s.bind(('127.0.0.1', 9999))
```

创建Socket时， `SOCK_DGRAM` 指定了这个Socket的类型是UDP。绑定端口和TCP一样，但是不需要调用 `listen()` 方法，而是直接接收来自任何客户端的数据：

```
print('Bind UDP on 9999...')
while True:
    # 接收数据:
    data, addr = s.recvfrom(1024)
    print('Received from %s:%s.' % addr)
    s.sendto(b'Hello, %s!' % data, addr)
```

`recvfrom()` 方法返回数据和客户端的地址与端口，这样，服务器收到数据后，直接调用 `sendto()` 就可以把数据用UDP发给客户端。

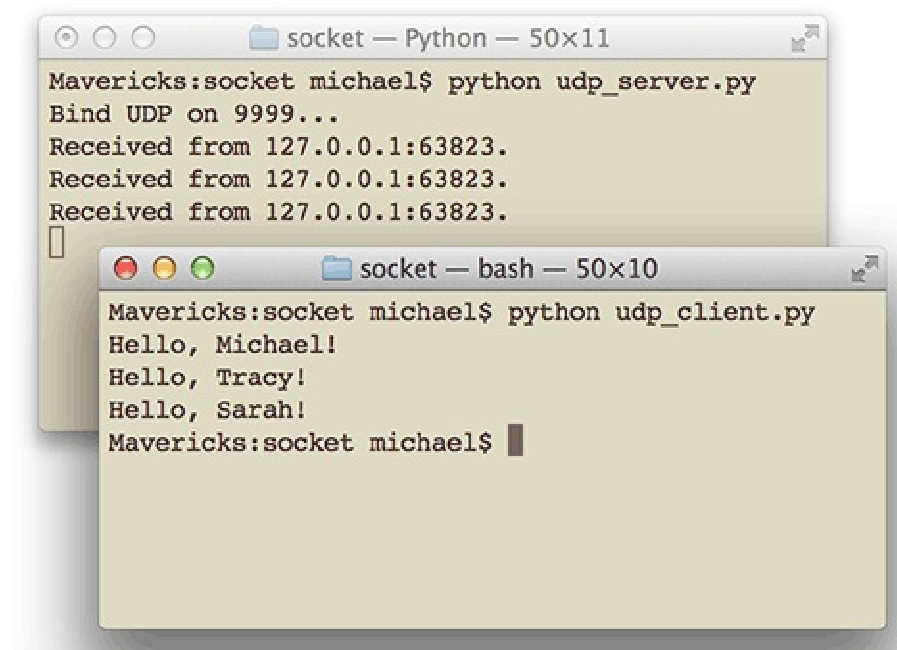
注意这里省掉了多线程，因为这个例子很简单。

客户端使用UDP时，首先仍然创建基于UDP的Socket，然后，不需要调用 `connect()` ，直接通过 `sendto()` 给服务器发数据：

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for data in [b'Michael', b'Tracy', b'Sarah']:
    # 发送数据:
    s.sendto(data, ('127.0.0.1', 9999))
    # 接收数据:
    print(s.recv(1024).decode('utf-8'))
s.close()
```

从服务器接收数据仍然调用 `recv()` 方法。

仍然用两个命令行分别启动服务器和客户端测试，结果如下：



小结

UDP的使用与TCP类似，但是不需要建立连接。此外，服务器绑定UDP端口和TCP端口互不冲突，也就是说，UDP的9999端口与TCP的9999端口可以各自绑定。

参考源码

udp_server.py

udp_client.py

电子邮件

Email的历史比Web还要久远，直到现在，Email也是互联网上应用非常广泛的服务。

几乎所有的编程语言都支持发送和接收电子邮件，但是，先等等，在我们开始编写代码之前，有必要搞清楚电子邮件是如何在互联网上运作的。

我们来看看传统邮件是如何运作的。假设你现在在北京，要给一个香港的朋友发一封信，怎么做呢？

首先你得写好信，装进信封，写上地址，贴上邮票，然后就近找个邮局，把信仍进去。

信件会从就近的小邮局转运到大邮局，再从大邮局往别的城市发，比如先发到天津，再走海运到达香港，也可能走京九线到香港，但是你不用关心具体路线，你只需要知道一件事，就是信件走得很慢，至少要几天时间。

信件到达香港的某个邮局，也不会直接送到朋友的家里，因为邮局的叔叔是很聪明的，他怕你的朋友不在家，一趟一趟地白跑，所以，信件会投递到你的朋友的邮箱里，邮箱可能在公寓的一层，或者家门口，直到你的朋友回家的时候检查邮箱，发现信件后，就可以取到邮件了。

电子邮件的流程基本上也是按上面的方式运作的，只不过速度不是按天算，而是按秒算。

现在我们回到电子邮件，假设我们自己的电子邮件地址是 `me@163.com` ，对方的电子邮件地址是 `friend@sina.com` （注意地址都是虚构的哈），现在我们用 `Outlook` 或者 `Foxmail` 之类的软件写好邮件，填上对方的Email地址，点“发送”，电子邮件就发出去了。这些电子邮件软件被称为**MUA**：Mail User Agent——邮件用户代理。

Email从MUA发出去，不是直接到达对方电脑，而是发到**MTA**：Mail Transfer Agent——邮件传输代理，就是那些Email服务提供商，比如网易、新浪等等。由于我们自己的电子邮件是 `163.com` ，所以，Email首先被投递到网易提供的MTA，再由网易的MTA发到对方服务商，也就是新浪的MTA。这个过程中间可能还会经过别的MTA，但是我们不关心具体路线，我们只关心速度。

Email到达新浪的MTA后，由于对方使用的是 `@sina.com` 的邮箱，因此，新浪的MTA会把Email投递到邮件的最终目的地**MDA**：Mail Delivery Agent——邮件投递代理。Email到达MDA后，就静静地躺在新浪的某个服务器上，存放在某个文件或特殊的数据库里，我们将这个长期保存邮件的地方称之为电子邮箱。

同普通邮件类似，Email不会直接到达对方的电脑，因为对方电脑不一定开机，开机也不一定联网。对方要取到邮件，必须通过MUA从MDA上把邮件取到自己的电脑上。

所以，一封电子邮件的旅程就是：

```
发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人
```

有了上述基本概念，要编写程序来发送和接收邮件，本质上就是：

1. 编写MUA把邮件发到MTA；
2. 编写MUA从MDA上收邮件。

发邮件时，MUA和MTA使用的协议就是SMTP：Simple Mail Transfer Protocol，后面的MTA到另一个MTA也是用SMTP协议。

收邮件时，MUA和MDA使用的协议有两种：POP：Post Office Protocol，目前版本是3，俗称POP3；IMAP：Internet Message Access Protocol，目前版本是4，优点是不但能取邮件，还可以直接操作MDA上存储的邮件，比如从收件箱移到垃圾箱，等等。

邮件客户端软件在发邮件时，会让你先配置SMTP服务器，也就是你要发到哪个MTA上。假设你正在使用163的邮箱，你就不能直接发到新浪的MTA上，因为它只服务新浪的用户，所以，你得填163提供的SMTP服务器地址： `smtp.163.com` ，为了证明你是163的用户，SMTP服务器还要求你填写邮箱地址和邮箱口令，这样，MUA才能正常地把Email通过SMTP协议发送到MTA。

类似的，从MDA收邮件时，MDA服务器也要求验证你的邮箱口令，确保不会有人冒充你收取你的邮件，所以，Outlook之类的邮件客户端会要求你填写POP3或IMAP服务器地址、邮箱地址和口令，这样，MUA才能顺利地通过POP或IMAP协议从MDA取到邮件。

在使用Python收发邮件前，请先准备好至少两个电子邮件，如 `xxx@163.com` ， `xxx@sina.com` ， `xxx@qq.com` 等，注意两个邮箱不要用同一家邮件服务商。

最后特别注意，目前大多数邮件服务商都需要手动打开SMTP发信和POP收信的功能，否则只允许在网页登录：



SMTP发送邮件

SMTP是发送邮件的协议，Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。

Python对SMTP支持有 `smtplib` 和 `email` 两个模块， `email` 负责构造邮件， `smtplib` 负责发送邮件。

首先，我们来构造一个最简单的纯文本邮件：

```
from email.mime.text import MIMEText
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
```

注意到构造 `MIMEText` 对象时，第一个参数就是邮件正文，第二个参数是MIME的subtype，传入 `'plain'` 表示纯文本，最终的MIME就是 `'text/plain'` ，最后一定要用 `utf-8` 编码保证多语言兼容性。

然后，通过SMTP发出去：

```
# 输入Email地址和口令:
from_addr = input('From: ')
password = input('Password: ')
# 输入收件人地址:
to_addr = input('To: ')
# 输入SMTP服务器地址:
smtp_server = input('SMTP server: ')

import smtplib
server = smtplib.SMTP(smtp_server, 25) # SMTP协议默认端口是25
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们用 `set_debuglevel(1)` 就可以打印出和SMTP服务器交互的所有信息。SMTP协议就是简单的文本命令和响应。 `login()` 方法用来登录SMTP服务器， `sendmail()` 方法就是发邮件，由于可以一次发给多个人，所以传入一

个 `list` ，邮件正文是一个 `str` ， `as_string()` 把 `MIMEText` 对象变成 `str` 。

如果一切顺利，就可以在收件人信箱中收到我们刚发送的Email：



hello, send by Python...

仔细观察，发现如下问题：

1. 邮件没有主题；
2. 收件人的名字没有显示为友好的名字，比如 `Mr Green <green@example.com>` ；
3. 明明收到了邮件，却提示不在收件人中。

这是因为邮件主题、如何显示发件人、收件人等信息并不是通过SMTP协议发给MTA，而是包含在发给MTA的文本中的，所以，我们必须把 `From` 、 `To` 和 `Subject` 添加到 `MIMEText` 中，才是一封完整的邮件：

```
from email import encoders
from email.header import Header
from email.mime.text import MIMEText
from email.utils import parseaddr, formataddr

import smtplib

def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

from_addr = input('From: ')
password = input('Password: ')
to_addr = input('To: ')
smtp_server = input('SMTP server: ')

msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')
msg['From'] = _format_addr('Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

server = smtplib.SMTP(smtp_server, 25)
server.set_debuglevel(1)
server.login(from_addr, password)
server.sendmail(from_addr, [to_addr], msg.as_string())
server.quit()
```

我们编写了一个函数 `_format_addr()` 来格式化一个邮件地址。注意不能简单地传入 `name <addr@example.com>` ，因为如果包含中文，需要通过 `Header` 对象进行编码。

`msg['To']` 接收的是字符串而不是list，如果有多个邮件地址，用 `,` 分隔即可。

再发送一遍邮件，就可以在收件人邮箱中看到正确的标题、发件人和收件人：

来自SMTP的问候……
发件人：Python爱好者 <xxxxxx@163.com>
时　间：2014年8月14日(星期四) 下午3:45
收件人：管理员 <xxxxxx@qq.com>

hello, send by Python...

你看到的收件人的名字很可能不是我们传入的 `管理员` ，因为很多邮件服务商在显示邮件时，会把收件人名字自动替换为用户注册的名字，但是其他收件人名字的显示不受影响。

如果我们查看Email的原始内容，可以看到如下经过编码的邮件头：

```
From: =?utf-8?b?UHl0aG9u54ix5aW96ICF?= <xxxxxx@163.com>
To: =?utf-8?b?566h55CG5ZGY?= <xxxxxx@qq.com>
Subject: =?utf-8?b?5p2l6IeqU01UUOeahOmXruWAmeKApuKApg==?=
```

这就是经过 `Header` 对象编码的文本，包含utf-8编码信息和Base64编码的文本。如果我们自己来手动构造这样的编码文本，显然比较复杂。

发送HTML邮件

如果我们要发送HTML邮件，而不是普通的纯文本文件怎么办？方法很简单，在构造 `MIMEText` 对象时，把HTML字符串传进去，再把第二个参数由 `plain` 变为 `html` 就可以了：

```
msg = MIMEText('<html><body><h1>Hello</h1>' +
    '<p>send by <a href="http://www.python.org">Python</a>...</p>'
    '</body></html>', 'html', 'utf-8')
```

再发送一遍邮件，你将看到以HTML显示的邮件：

来自SMTP的问候......

发件人：Python爱好者 <xxxxxx@163.com>

时　间：2014年8月14日(星期四) 下午4:06

收件人：管理员 <xxxxxx@qq.com>

Hello

send by Python...

发送附件

如果Email中要加上附件怎么办？带附件的邮件可以看做包含若干部分的邮件：文本和各个附件本身，所以，可以构造一个 `MIMEMultipart` 对象代表邮件本身，然后往里面加上一个 `MIMEText` 作为邮件正文，再继续往里面加上表示附件的 `MIMEBase` 对象即可：

```
# 邮件对象:
msg = MIMEMultipart()
msg['From'] = _format_addr('Python爱好者 <%s>' % from_addr)
msg['To'] = _format_addr('管理员 <%s>' % to_addr)
msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()

# 邮件正文是MIMEText:
msg.attach(MIMEText('send with file...', 'plain', 'utf-8'))

# 添加附件就是加上一个MIMEBase，从本地读取一个图片:
with open('/Users/michael/Downloads/test.png', 'rb') as f:
    # 设置附件的MIME和文件名，这里是png类型:
    mime = MIMEBase('image', 'png', filename='test.png')
    # 加上必要的头信息:
    mime.add_header('Content-Disposition', 'attachment', filename=
    mime.add_header('Content-ID', '<0>')
    mime.add_header('X-Attachment-Id', '0')
    # 把附件的内容读进来:
    mime.set_payload(f.read())
    # 用Base64编码:
    encoders.encode_base64(mime)
    # 添加到MIMEMultipart:
    msg.attach(mime)
```

然后，按正常发送流程把 `msg` （注意类型已变为 `MIMEMultipart` ）发送出去，就可以收到如下带附件的邮件：

来自SMTP的问候......

发件人：Python爱好者 <xxxxxx@163.com>

时　间：2014年8月14日(星期四) 下午5:08

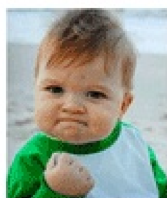
收件人：管理员 <xxxxxx@qq.com>

附　件：1 个 (test.png)

send with file...

附件(1 个)

普通附件



test.png (80.13K)

下载 预览 收藏 转存

发送图片

如果要把一个图片嵌入到邮件正文中怎么做？直接在HTML邮件中链接图片地址行不行？答案是，大部分邮件服务商都会自动屏蔽带有外链的图片，因为不知道这些链接是否指向恶意网站。

要把图片嵌入到邮件正文中，我们只需按照发送附件的方式，先把邮件作为附件添加进去，然后，在HTML中通过引用 `src="cid:0"` 就可以把附件作为图片嵌入了。如果有多个图片，给它们依次编号，然后引用不同的 `cid:x` 即可。

把上面代码加入 `MIMEMultipart` 的 `MIMEText` 从 `plain` 改为 `html` ，然后在适当的位置引用图片：

```
msg.attach(MIMEText('<html><body><h1>Hello</h1>' +
    '<p><img src="cid:0"></p>' +
    '</body></html>', 'html', 'utf-8'))
```

再次发送，就可以看到图片直接嵌入到邮件正文的效果：

来自SMTP的问候……

发件人：Python爱好者 <asklxf@163.com>

时　间：2014年8月14日(星期四) 下午5:27

收件人：Xuefeng <18224514@qq.com>

Hello



同时支持HTML和Plain格式

如果我们发送HTML邮件，收件人通过浏览器或者Outlook之类的软件是可以正常浏览邮件内容的，但是，如果收件人使用的设备太古老，查看不了HTML邮件怎么办？

办法是在发送HTML的同时再附加一个纯文本，如果收件人无法查看HTML格式的邮件，就可以自动降级查看纯文本邮件。

利用 `MIMEMultipart` 就可以组合一个HTML和Plain，要注意指定subtype是 `alternative` ：

```
msg = MIMEMultipart('alternative')
msg['From'] = ...
msg['To'] = ...
msg['Subject'] = ...

msg.attach(MIMEText('hello', 'plain', 'utf-8'))
msg.attach(MIMEText('<html><body><h1>Hello</h1></body></html>', 'ht
# 正常发送msg对象...
```

加密SMTP

使用标准的25端口连接SMTP服务器时，使用的是明文传输，发送邮件的整个过程可能会被窃听。要更安全地发送邮件，可以加密SMTP会话，实际上就是先创建SSL安全连接，然后再使用SMTP协议发送邮件。

某些邮件服务商，例如Gmail，提供的SMTP服务必须要加密传输。我们来看看如何通过Gmail提供的安全SMTP发送邮件。

必须知道，Gmail的SMTP端口是587，因此，修改代码如下：

```
smtp_server = 'smtp.gmail.com'
smtp_port = 587
server = smtplib.SMTP(smtp_server, smtp_port)
server.starttls()
# 剩下的代码和前面的一模一样:
server.set_debuglevel(1)
...
```

只需要在创建 `SMTP` 对象后，立刻调用 `starttls()` 方法，就创建了安全连接。后面的代码和前面的发送邮件代码完全一样。

如果因为网络问题无法连接Gmail的SMTP服务器，请相信我们的代码是没有问题的，你需要对你的网络设置做必要的调整。

小结

使用Python的smtplib发送邮件十分简单，只要掌握了各种邮件类型的构造方法，正确设置好邮件头，就可以顺利发出。

构造一个邮件对象就是一个 `Messag` 对象，如果构造一个 `MIMEText` 对象，就表示一个文本邮件对象，如果构造一个 `MIMEImage` 对象，就表示一个作为附件的图片，要把多个对象组合起来，就用 `MIMEMultipart` 对象，而 `MIMEBase` 可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
   +- MIMEMultipart
   +- MIMENonMultipart
      +- MIMEMessage
      +- MIMEText
      +- MIMEImage
```

这种嵌套关系就可以构造出任意复杂的邮件。你可以通过email.mime文档查看它们所在的包以及详细的用法。

参考源码

send_mail.py

POP3收取邮件

SMTP用于发送邮件，如果要收取邮件呢？

收取邮件就是编写一个**MUA**作为客户端，从**MDA**把邮件获取到用户的电脑或者手机上。收取邮件最常用的协议是**POP**协议，目前版本号是3，俗称**POP3**。

Python内置一个`poplib`模块，实现了POP3协议，可以直接用来收邮件。

注意到POP3协议收取的不是一个已经可以阅读的邮件本身，而是邮件的原始文本，这和SMTP协议很像，SMTP发送的也是经过编码后的一大段文本。

要把POP3收取的文本变成可以阅读的邮件，还需要用`email`模块提供的各种类来解析原始文本，变成可阅读的邮件对象。

所以，收取邮件分两步：

第一步：用`poplib`把邮件的原始文本下载到本地；

第二部：用`email`解析原始文本，还原为邮件对象。

通过POP3下载邮件

POP3协议本身很简单，以下面的代码为例，我们来获取最新的一封邮件内容：

```
import poplib

# 输入邮件地址, 口令和POP3服务器地址:
email = input('Email: ')
password = input('Password: ')
pop3_server = input('POP3 server: ')

# 连接到POP3服务器:
server = poplib.POP3(pop3_server)
# 可以打开或关闭调试信息:
server.set_debuglevel(1)
# 可选:打印POP3服务器的欢迎文字:
print(server.getwelcome().decode('utf-8'))

# 身份认证:
server.user(email)
server.pass_(password)

# stat()返回邮件数量和占用空间:
print('Messages: %s. Size: %s' % server.stat())
# list()返回所有邮件的编号:
resp, mails, octets = server.list()
# 可以查看返回的列表类似[b'1 82923', b'2 2184', ...]
print(mails)

# 获取最新一封邮件, 注意索引号从1开始:
index = len(mails)
resp, lines, octets = server.retr(index)

# lines存储了邮件的原始文本的每一行,
# 可以获得整个邮件的原始文本:
msg_content = b'\r\n'.join(lines).decode('utf-8')
# 稍后解析出邮件:
msg = Parser().parsestr(msg_content)

# 可以根据邮件索引号直接从服务器删除邮件:
# server.dele(index)
# 关闭连接:
server.quit()
```

用POP3获取邮件其实很简单，要获取所有邮件，只需要循环使用 `retr()` 把每一封邮件内容拿到即可。真正麻烦的是把邮件的原始内容解析为可以阅读的邮件对象。

解析邮件

解析邮件的过程和上一节构造邮件正好相反，因此，先导入必要的模块：

```
from email.parser import Parser
from email.header import decode_header
from email.utils import parseaddr

import poplib
```

只需要一行代码就可以把邮件内容解析为 `Message` 对象：

```
msg = Parser().parsestr(msg_content)
```

但是这个 `Message` 对象本身可能是一个 `MIMEMultipart` 对象，即包含嵌套的其他 `MIMEBase` 对象，嵌套可能还不止一层。

所以我们要递归地打印出 `Message` 对象的层次结构：

```
# indent用于缩进显示:
def print_info(msg, indent=0):
    if indent == 0:
        for header in ['From', 'To', 'Subject']:
            value = msg.get(header, '')
            if value:
                if header=='Subject':
                    value = decode_str(value)
                else:
                    hdr, addr = parseaddr(value)
                    name = decode_str(hdr)
                    value = u'%s <%s>' % (name, addr)
            print('%s%s: %s' % ('  ' * indent, header, value))
    if (msg.is_multipart()):
        parts = msg.get_payload()
        for n, part in enumerate(parts):
            print('%spart %s' % ('  ' * indent, n))
            print('%s--------------------' % ('  ' * indent))
            print_info(part, indent + 1)
    else:
        content_type = msg.get_content_type()
        if content_type=='text/plain' or content_type=='text/html'
            content = msg.get_payload(decode=True)
            charset = guess_charset(msg)
            if charset:
                content = content.decode(charset)
            print('%sText: %s' % ('  ' * indent, content + '...'))
        else:
            print('%sAttachment: %s' % ('  ' * indent, content_type
```

邮件的Subject或者Email中包含的名字都是经过编码后的str，要正常显示，就必须decode：

```
def decode_str(s):
    value, charset = decode_header(s)[0]
    if charset:
        value = value.decode(charset)
    return value
```

`decode_header()` 返回一个list，因为像 `Cc` 、 `Bcc` 这样的字段可能包含多个邮件地址，所以解析出来的会有多个元素。上面的代码我们偷了个懒，只取了第一个元素。

文本邮件的内容也是str，还需要检测编码，否则，非UTF-8编码的邮件都无法正常显示：

```
def guess_charset(msg):
    charset = msg.get_charset()
    if charset is None:
        content_type = msg.get('Content-Type', '').lower()
        pos = content_type.find('charset=')
        if pos >= 0:
            charset = content_type[pos + 8:].strip()
    return charset
```

把上面的代码整理好，我们就可以来试试收取一封邮件。先往自己的邮箱发一封邮件，然后用浏览器登录邮箱，看看邮件收到没，如果收到了，我们就来用Python程序把它收到本地：



运行程序，结果如下：

```
+OK Welcome to coremail Mail Pop3 Server (163coms[...])
Messages: 126\. Size: 27228317

From: Test <xxxxxx@qq.com>
To: Python爱好者 <xxxxxx@163.com>
Subject: 用POP3收取邮件
part 0
--------------------
  part 0
  --------------------
    Text: Python可以使用POP3收取邮件……...
  part 1
  --------------------
    Text: Python可以<a href="...">使用POP3</a>收取邮件……...
part 1
--------------------
  Attachment: application/octet-stream
```

我们从打印的结构可以看出，这封邮件是一个 `MIMEMultipart` ，它包含两部分：第一部分又是一个 `MIMEMultipart` ，第二部分是一个附件。而内嵌的 `MIMEMultipart` 是一个 `alternative` 类型，它包含一个纯文本格式的 `MIMEText` 和一个HTML格式的 `MIMEText` 。

小结

用Python的 `poplib` 模块收取邮件分两步：第一步是用POP3协议把邮件获取到本地，第二步是用 `email` 模块把原始邮件解析为 `Message` 对象，然后，用适当的形式把邮件内容展示给用户即可。

参考源码

fetch_mail.py

访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用 `,` 隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name":"Michael","score":99},
    {"name":"Bob","score":85},
    {"name":"Bart","score":59},
    {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

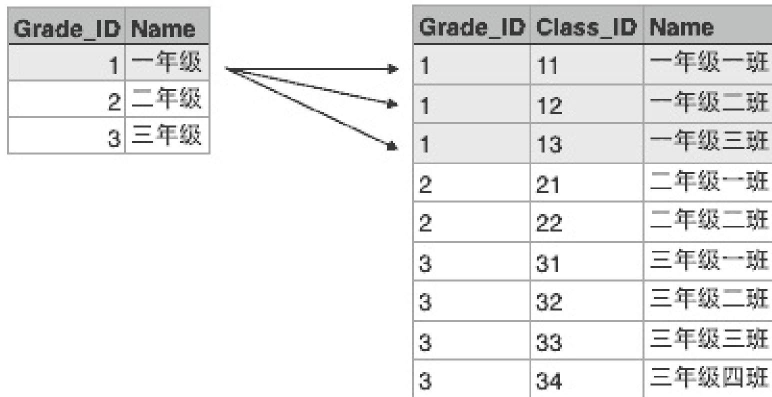
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



Grade_ID	Name
1	一年级
2	二年级
3	三年级

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

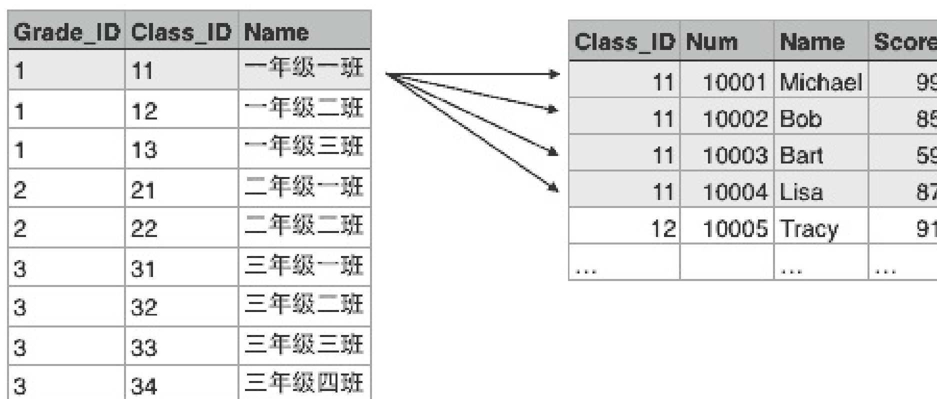
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
---------+----------+----------
grade_id | class_id | name
---------+----------+----------
1        | 11       | 一年级一班
---------+----------+----------
1        | 12       | 一年级二班
---------+----------+----------
1        | 13       | 一年级三班
---------+----------+----------
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...		...	...

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，推荐Coursera课程：

英文：https://www.coursera.org/course/db

中文：http://c.open.163.com/coursera/courseIntro.htm?cid=12

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；
- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为Python开发工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

为了能继续后面的学习，你需要从MySQL官方网站下载并安装MySQL Community Server 5.6，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。

使用SQLite

SQLite是一种嵌入式数据库，它的数据库就是一个文件。由于SQLite本身是C写的，而且体积很小，所以，经常被集成到各种应用程序中，甚至在iOS和Android的App中都可以集成。

Python就内置了SQLite3，所以，在Python中使用SQLite，不需要安装任何东西，直接使用。

在使用SQLite前，我们先要搞清楚几个概念：

表是数据库中存放关系数据的集合，一个数据库里面通常都包含多个表，比如学生的表，班级的表，学校的表，等等。表和表之间通过外键关联。

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；

连接到数据库后，需要打开游标，称之为Cursor，通过Cursor执行SQL语句，然后，获得执行结果。

Python定义了一套操作数据库的API接口，任何数据库要连接到Python，只需要提供符合Python标准的数据库驱动即可。

由于SQLite的驱动内置在Python标准库中，所以我们可以直接来操作SQLite数据库。

我们在Python交互式命令行实践一下：

```
# 导入SQLite驱动:
>>> import sqlite3
# 连接到SQLite数据库
# 数据库文件是test.db
# 如果文件不存在，会自动在当前目录创建:
>>> conn = sqlite3.connect('test.db')
# 创建一个Cursor:
>>> cursor = conn.cursor()
# 执行一条SQL语句，创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key,
<sqlite3.Cursor object at 0x10f8aa260>
# 继续执行一条SQL语句，插入一条记录:
>>> cursor.execute('insert into user (id, name) values (\'1\', \'Mi
<sqlite3.Cursor object at 0x10f8aa260>
# 通过rowcount获得插入的行数:
>>> cursor.rowcount
1
# 关闭Cursor:
>>> cursor.close()
# 提交事务:
>>> conn.commit()
# 关闭Connection:
>>> conn.close()
```

我们再试试查询记录：

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
# 执行查询语句:
>>> cursor.execute('select * from user where id=?', '1')
<sqlite3.Cursor object at 0x10f8aa340>
# 获得查询结果集:
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
>>> cursor.close()
>>> conn.close()
```

使用Python的DB-API时，只要搞清楚 `Connection` 和 `Cursor` 对象，打开后一定记得关闭，就可以放心地使用。

使用 `Cursor` 对象执行 `insert` ， `update` ， `delete` 语句时，执行结果由 `rowcount` 返回影响的行数，就可以拿到执行结果。

使用 `Cursor` 对象执行 `select` 语句时，通过 `featchall()` 可以拿到结果集。结果集是一个list，每个元素都是一个tuple，对应一行记录。

如果SQL语句带有参数，那么需要把参数按照位置传递给 `execute()` 方法，有几个 `?` 占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where id=?', '1')
```

SQLite支持常见的标准SQL语句以及几种常见的数据类型。具体文档请参阅SQLite官方网站。

小结

在Python中操作数据库时，要先导入数据库对应的驱动，然后，通过 `Connection` 对象和 `Cursor` 对象操作数据。

要确保打开的 `Connection` 对象和 `Cursor` 对象都正确地被关闭，否则，资源就会泄露。

如何才能确保出错的情况下也关闭掉 `Connection` 对象和 `Cursor` 对象呢？请回忆 `try:...except:...finally:...` 的用法。

练习

请编写函数，在Sqlite中根据分数段查找指定的名字：

```
# -*- coding: utf-8 -*-

import os, sqlite3

db_file = os.path.join(os.path.dirname(__file__), 'test.db')
if os.path.isfile(db_file):
    os.remove(db_file)

# 初始数据:
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
cursor.execute('create table user(id varchar(20) primary key, name
cursor.execute(r"insert into user values ('A-001', 'Adam', 95)")
cursor.execute(r"insert into user values ('A-002', 'Bart', 62)")
cursor.execute(r"insert into user values ('A-003', 'Lisa', 78)")
cursor.close()
conn.commit()
conn.close()

def get_score_in(low, high):
    ' 返回指定分数区间的名字，按分数从低到高排序 '

    pass

# 测试:
assert get_score_in(80, 95) == ['Adam'], get_score_in(80, 95)
assert get_score_in(60, 80) == ['Bart', 'Lisa'], get_score_in(60, 8
assert get_score_in(60, 100) == ['Bart', 'Lisa', 'Adam'], get_score

print('Pass')
```

参考源码

do_sqlite.py

使用MySQL

MySQL是Web世界中使用最广泛的数据库服务器。SQLite的特点是轻量级、可嵌入，但不能承受高并发访问，适合桌面和移动应用。而MySQL是为服务器端设计的数据库，能承受高并发访问，同时占用的内存也远远大于SQLite。

此外，MySQL内部有多种数据库引擎，最常用的引擎是支持数据库事务的InnoDB。

安装MySQL

可以直接从MySQL官方网站下载最新的Community Server 5.6.x版本。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password` 。

在Windows上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf` ：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+--------------------------+----------------------------------------
| Variable_name            | Value
+--------------------------+----------------------------------------
| character_set_client     | utf8
| character_set_connection | utf8
| character_set_database   | utf8
| character_set_filesystem | binary
| character_set_results    | utf8
| character_set_server     | utf8
| character_set_system     | utf8
| character_sets_dir       | /usr/local/mysql-5.1.65-osx10.6-x86_64
+--------------------------+----------------------------------------
8 rows in set (0.00 sec)
```

看到 `utf8` 字样就表示编码设置正确。

安装MySQL驱动

由于MySQL服务器以独立的进程运行，并通过网络对外服务，所以，需要支持Python的MySQL驱动来连接到MySQL服务器。MySQL官方提供了mysql-connector-python驱动，但是安装的时候需要给pip命令加上参数 `--allow-external` ：

```
$ pip install mysql-connector-python --allow-external mysql-connect
```

我们演示如何连接到MySQL服务器的test数据库：

```
# 导入MySQL驱动:
>>> import mysql.connector
# 注意把password设为你的root口令:
>>> conn = mysql.connector.connect(user='root', password='password
>>> cursor = conn.cursor()
# 创建user表:
>>> cursor.execute('create table user (id varchar(20) primary key,
# 插入一行记录，注意MySQL的占位符是%s:
>>> cursor.execute('insert into user (id, name) values (%s, %s)', [
>>> cursor.rowcount
1
# 提交事务:
>>> conn.commit()
>>> cursor.close()
# 运行查询:
>>> cursor = conn.cursor()
>>> cursor.execute('select * from user where id = %s', ['1'])
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
# 关闭Cursor和Connection:
>>> cursor.close()
True
>>> conn.close()
```

由于Python的DB-API定义都是通用的，所以，操作MySQL的数据库代码和SQLite类似。

小结

- 执行INSERT等操作后要调用 `commit()` 提交事务；
- MySQL的SQL占位符是 `%s` 。

参考源码

do_mysql.py

使用SQLAlchemy

数据库表是一个二维表，包含多行多列。把一个表的内容用Python的数据结构表示出来的话，可以用一个list表示多行，list的每一个元素是tuple，表示一行记录，比如，包含 `id` 和 `name` 的 `user` 表：

```
[
    ('1', 'Michael'),
    ('2', 'Bob'),
    ('3', 'Adam')
]
```

Python的DB-API返回的数据结构就是像上面这样表示的。

但是用tuple表示一行很难看出表的结构。如果把一个tuple用class实例来表示，就可以更容易地看出表的结构来：

```
class User(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

[
    User('1', 'Michael'),
    User('2', 'Bob'),
    User('3', 'Adam')
]
```

这就是传说中的ORM技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

在Python中，最有名的ORM框架是SQLAlchemy。我们来看看SQLAlchemy的用法。

首先通过pip安装SQLAlchemy：

```
$ pip install sqlalchemy
```

然后，利用上次我们在MySQL的test数据库中创建的 `user` 表，用SQLAlchemy来试试：

第一步，导入SQLAlchemy，并初始化DBSession：

```
# 导入:
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# 创建对象的基类:
Base = declarative_base()

# 定义User对象:
class User(Base):
    # 表的名字:
    __tablename__ = 'user'

    # 表的结构:
    id = Column(String(20), primary_key=True)
    name = Column(String(20))

# 初始化数据库连接:
engine = create_engine('mysql+mysqlconnector://root:password@localh
# 创建DBSession类型:
DBSession = sessionmaker(bind=engine)
```

以上代码完成SQLAlchemy的初始化和具体每个表的class定义。如果有多个表，就继续定义其他class，例如School：

```
class School(Base):
    __tablename__ = 'school'
    id = ...
    name = ...
```

`create_engine()` 用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息：

```
'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
```

你只需要根据需要替换掉用户名、口令等信息即可。

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，我们向数据库表中添加一行记录，可以视为添加一个 `User` 对象：

```
# 创建session对象:
session = DBSession()
# 创建新User对象:
new_user = User(id='5', name='Bob')
# 添加到session:
session.add(new_user)
# 提交即保存到数据库:
session.commit()
# 关闭session:
session.close()
```

可见，关键是获取session，然后把对象添加到session，最后提交并关闭。 `DBSession` 对象可视为当前数据库连接。

如何从数据库表中查询数据呢？有了ORM，查询出来的可以不再是tuple，而是 `User` 对象。SQLAlchemy提供的查询接口如下：

```
# 创建Session:
session = DBSession()
# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的name属性:
print('type:', type(user))
print('name:', user.name)
# 关闭Session:
session.close()
```

运行结果如下：

```
type: <class '__main__.User'>
name: Bob
```

可见，ORM就是把数据库表的行与相应的对象建立关联，互相转换。

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。

例如，如果一个User拥有多个Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')

class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # “多”的一方的book表是通过外键关联到user表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

当我们查询一个User对象时，该对象的books属性将返回一个包含若干个Book对象的list。

小结

ORM框架的作用就是把数据库表的一行记录与一个对象互相做自动转换。

正确使用ORM的前提是了解关系数据库的原理。

参考源码

do_sqlalchemy.py

Web开发

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登陆到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

1. 静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；
2. CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。
3. ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。

4. MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

Python的诞生历史比Web还要早，由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。

Python有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

本章我们会详细讨论Python Web开发技术。

HTTP协议简介

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器之间的传输协议是HTTP，所以：

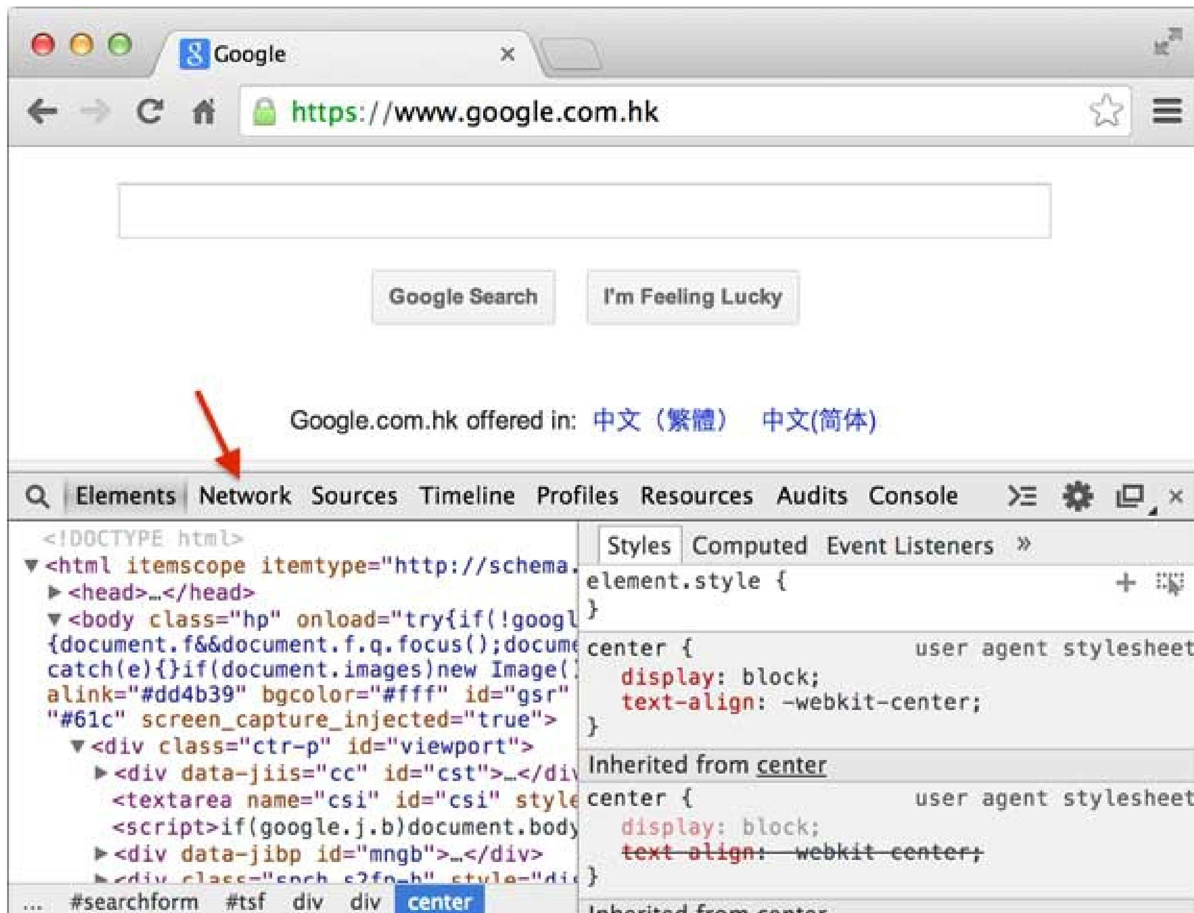
- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

在举例子之前，我们需要安装Google的Chrome浏览器。

为什么要使用Chrome浏览器而不是IE呢？因为IE实在是太慢了，并且，IE对于开发和调试Web应用程序完全是一点用也没有。

我们需要在浏览器很方便地调试我们的Web应用，而Chrome提供了一套完整地调试工具，非常适合Web开发。

安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



`Elements` 显示网页的结构， `Network` 显示浏览器和服务器的通信。我们点 `Network` ，确保第一个小红灯亮着，Chrome就会记录所有浏览器和服务器之间的通信：



当我们在地址栏输入 `www.sina.com.cn` 时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过 `Network` 的记录，我们就可以知道。在 `Network` 中，定位到第一条记录，点击，右侧将显示 `Request Headers` ，点击右侧的 `view source` ，我们就可以看到浏览器发给新浪服务器的请求：

```
Remote Address: 202.108.33.60:80
Request URL: http://www.sina.com.cn/
Request Method: GET
Status Code: 200 OK
Request Headers    view parsed
GET / HTTP/1.1
Host: www.sina.com.cn
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application
/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chro
me/34.0.1847.131 Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,en-US;q=0.
4,zh-TW;q=0.2
Cookie: UOR=blog.sina.com.cn,blog.sina.com.cn,; U_T
```

最主要的头两行分析如下，第一行：

```
GET / HTTP/1.1
```

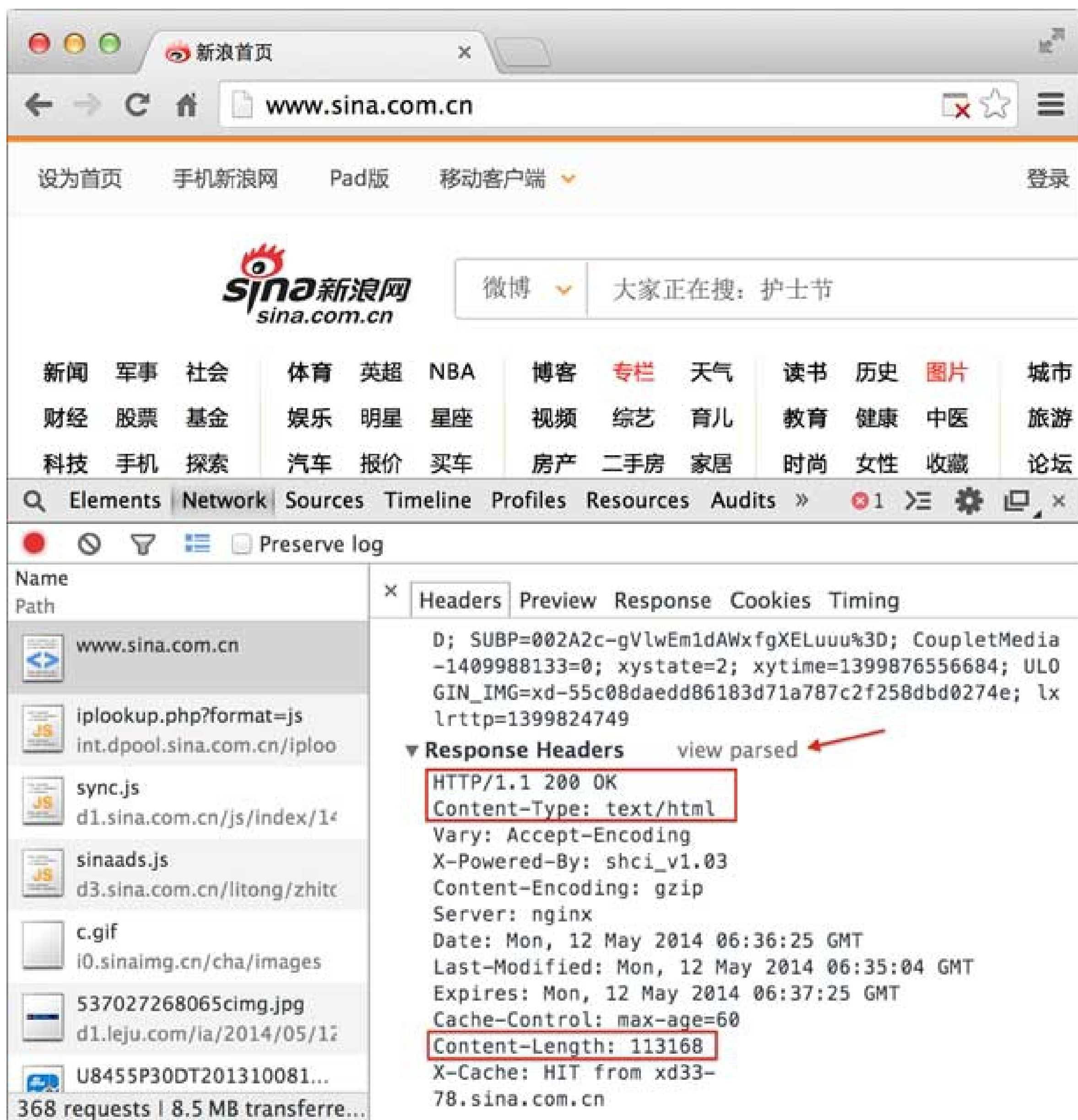
`GET` 表示一个读取请求，将从服务器获得网页数据， `/` 表示URL的路径，URL总是以 `/` 开头， `/` 就表示首页，最后的 `HTTP/1.1` 指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用一个TCP连接，以加快传输速度。

从第二行开始，每一行都类似于 `Xxx: abcdefg` ：

```
Host: www.sina.com.cn
```

表示请求的域名是 `www.sina.com.cn` 。如果一台服务器有多个网站，服务器就需要通过 `Host` 来区分浏览器请求的是哪个网站。

继续往下找到 `Response Headers` ，点击 `view source` ，显示服务器返回的原始响应数据：



HTTP响应分为Header和Body两部分（Body是可选项），我们在 `Network` 中看到的Header最重要的几行如下：

```
200 OK
```

`200` 表示一个成功的响应，后面的 `OK` 是说明。失败的响应有 `404 Not Found` ：网页不存在， `500 Internal Server Error` ：服务器内部出错，等等。

```
Content-Type: text/html
```

`Content-Type` 指示响应的内容，这里是 `text/html` 表示HTML网页。请注意，浏览器就是依靠 `Content-Type` 来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠URL来判断响应的内容，所以，即使URL是 `http://example.com/abc.jpg` ，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：

```
<!DOCTYPE html>
<!--[30,131,1] published at 2014-05-12 14:35:02 from #153 by 9018-->
<html>
<head>
    <meta http-equiv="Content-type" content="text/html; charset=gb2312" />
    <title>新浪首页</title>

        <meta name="keywords" content="新浪,新浪网,SINA,sina,sina.com.cn,新浪首页,门户,资讯" />
        <meta name="description" content="新浪网为全球用户24小时提供全面及时的中文资讯，内容覆盖国内外突发新闻事件、体坛赛事、娱乐时尚、产业资讯、实用信息等，设有新闻、体育、娱乐、财经、科技、房产、汽车等30多个内容频道，同时开设博客、视频、论坛等自由互动交流空间。" />
        <meta name="stencil" content="PGLS000022" />
        <meta name="publishid" content="30,131,1" />
        <meta name="verify-v1" content="6HtwmypggdgP1NLw7NOuQBI2TW8+CfkYCoyeB8IDbn8=" />
        <meta name="360-site-verification" content="63349a2167ca11f4b9bd9a8d48354541" />
        <meta name="application-name" content="新浪首页"/>
        <meta name ="msapplication-TileImage" content="http://i1.sinaimg.cn/dy/deco/2013/0312/logo.png"/>
        <meta name="msapplication-TileColor" content="#ffbf27"/>
        <meta name="sogou_site_verification" content="BVIdHxKGrl"/>
<link rel="apple-touch-icon" href="http://i3.sinaimg.cn/home/2013/0331/U586P30DT20130331093840.png" />

        <script type="text/javascript">
    //js异步加载管理
    (function(){var w=this,d=document,version='1.0.7',data={},length=0,cbkLen=0;if(w.jsLoader){if(w.jsLoader.version>=version){return};data=w.jsLoader.getData();length=data.length};var addEvent=function(obj,eventType,func){if(obj.attachEvent){obj.attachEvent("on"+eventType,func)}else{obj.addEventListener(eventType,func,false)}};var domReady=false,ondomReady=function()
```

当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在 `Network` 下面能看到很多额外的HTTP请求。

HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: www.sina.com.cn

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据。

步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是 `http://www.sina.com.cn/` 的首页，但是新浪在HTML中可以链入其他服务器的资源，比如 `<img src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">`，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。HTTP GET请求的格式：

```
GET /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3
```

每个Header一行一个，换行符是 `\r\n` 。

HTTP POST请求的格式：

```
POST /path HTTP/1.1
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

当遇到连续两个 `\r\n` 时，Header部分结束，后面的数据全部是Body。

HTTP响应的格式：

```
200 OK
Header1: Value1
Header2: Value2
Header3: Value3

body data goes here...
```

HTTP响应如果包含body，也是通过 `\r\n\r\n` 来分隔的。请再次注意，Body的数据类型由 `Content-Type` 头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在 `Content-Encoding` 时，Body数据是被压缩的，最常见的压缩方式是gzip，所以，看到 `Content-Encoding: gzip` 时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

要详细了解HTTP协议，推荐“HTTP: The Definitive Guide”一书，非常不错，有中文译本：

HTTP权威指南

HTML简介

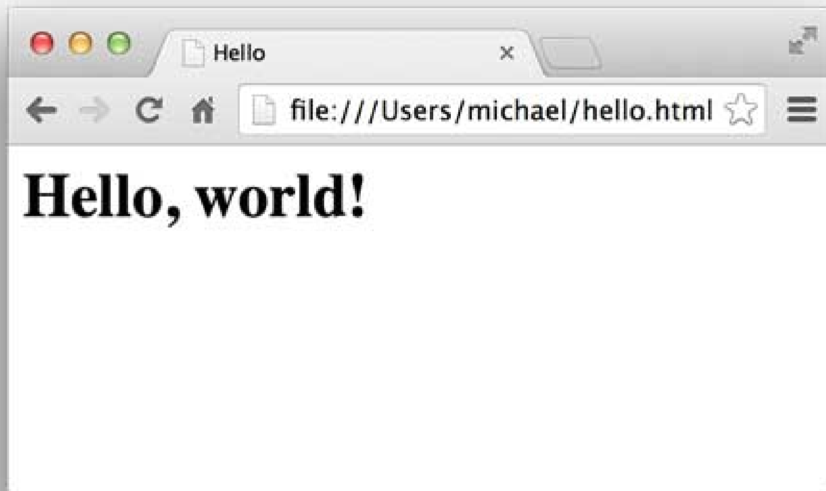
网页就是HTML？这么理解大概没错。因为网页中不但包含文字，还有图片、视频、Flash小游戏，有复杂的排版、动画效果，所以，HTML定义了一套语法规则，来告诉浏览器如何把一个丰富多彩的页面显示出来。

HTML长什么样？上次我们看了新浪首页的HTML源码，如果仔细数数，竟然有6000多行！

所以，学HTML，就不要指望从新浪入手了。我们来看看最简单的HTML长什么样：

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

可以用文本编辑器编写HTML，然后保存为 `hello.html` ，双击或者把文件拖到浏览器中，就可以看到效果：



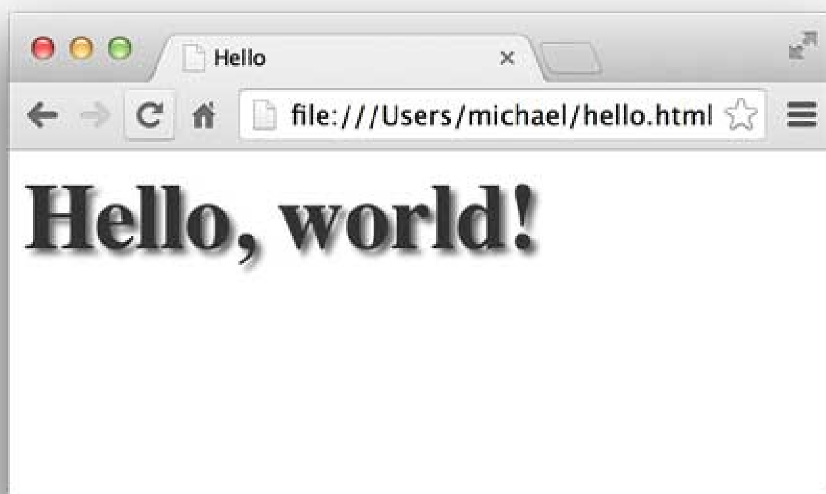
HTML文档就是一系列的Tag组成，最外层的Tag是 `<html>` 。规范的HTML也包含 `<head>...</head>` 和 `<body>...</body>` （注意不要和HTTP的Header、Body搞混了），由于HTML是富文档模型，所以，还有一系列的Tag用来表示链接、图片、表格、表单等等。

CSS简介

CSS是Cascading Style Sheets（层叠样式表）的简称，CSS用来控制HTML里的所有元素如何展现，比如，给标题元素 `<h1>` 加一个样式，变成48号字体，灰色，带阴影：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
</head>
<body>
  <h1>Hello, world!</h1>
</body>
</html>
```

效果如下：



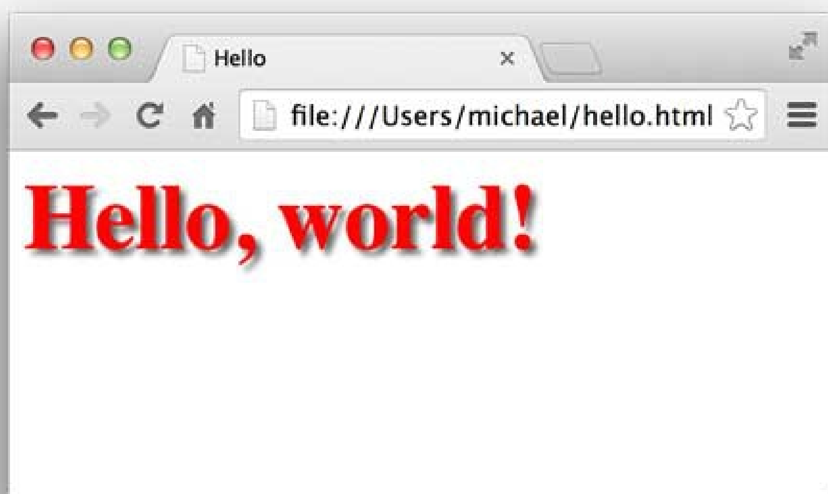
JavaScript简介

JavaScript虽然名称有个Java，但它和Java真的一点关系没有。JavaScript是为了让HTML具有交互性而作为脚本语言添加的，JavaScript既可以内嵌到HTML中，也可以从外部链接到HTML中。如果我们希望当用户点击标题时把标题变成红色，就

必须通过JavaScript来实现：

```
<html>
<head>
  <title>Hello</title>
  <style>
    h1 {
      color: #333333;
      font-size: 48px;
      text-shadow: 3px 3px 3px #666666;
    }
  </style>
  <script>
    function change() {
      document.getElementsByTagName('h1')[0].style.color = '#ff0000
    }
  </script>
</head>
<body>
  <h1 onclick="change()">Hello, world!</h1>
</body>
</html>
```

点击标题后效果如下：



小结

如果要学习Web开发，首先要对HTML、CSS和JavaScript作一定的了解。HTML定义了页面的内容，CSS来控制页面元素的样式，而JavaScript负责页面的交互逻辑。

讲解HTML、CSS和JavaScript就可以写3本书，对于优秀的Web开发人员来说，精通HTML、CSS和JavaScript是必须的，这里推荐一个在线学习网站w3schools：

http://www.w3schools.com/

以及一个对应的中文版本：

http://www.w3school.com.cn/

当我们用Python或者其他语言开发Web应用时，我们就是要在服务器端动态创建出HTML，这样，浏览器就会向不同的用户显示出不同的Web页面。

WSGI接口

了解了HTTP协议和HTML文档，我们其实就明白了一个Web应用的本质就是：

1. 浏览器发送一个HTTP请求；
2. 服务器收到请求，生成一个HTML文档；
3. 服务器把HTML文档作为HTTP响应的Body发送给浏览器；
4. 浏览器收到HTTP响应，从HTTP Body取出HTML文档并显示。

所以，最简单的Web应用就是先把HTML用文件保存好，用一个现成的HTTP服务器软件，接收用户请求，从文件中读取HTML，返回。Apache、Nginx、Lighttpd等这些常见的静态服务器就是干这件事情的。

如果要动态生成HTML，就需要把上述步骤自己来实现。不过，接受HTTP请求、解析HTTP请求、发送HTTP响应都是苦力活，如果我们自己来写这些底层代码，还没开始写动态HTML呢，就得花个把月去读HTTP规范。

正确的做法是底层代码由专门的服务器软件实现，我们用Python专注于生成HTML文档。因为我们不希望接触到TCP连接、HTTP原始请求和响应格式，所以，需要一个统一的接口，让我们专心用Python编写Web业务。

这个接口就是WSGI：Web Server Gateway Interface。

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello, web!”：

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']
```

上面的`application()`函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

- environ：一个包含所有HTTP请求信息的`dict`对象；
- start_response：一个发送HTTP响应的函数。

在 `application()` 函数中，调用：

```
start_response('200 OK', [('Content-Type', 'text/html')])
```

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次 `start_response()` 函数。 `start_response()` 函数接收两个参数，一个是HTTP响应码，一个是一组 `list` 表示的HTTP Header，每个Header用一个包含两个 `str` 的 `tuple` 表示。

通常情况下，都应该把 `Content-Type` 头发送给浏览器。其他很多常用的HTTP Header也应该发送。

然后，函数的返回值 `b'<h1>Hello, web!</h1>'` 将作为HTTP响应的Body发送给浏览器。

有了WSGI，我们关心的就是如何从 `environ` 这个 `dict` 对象拿到HTTP请求信息，然后构造HTML，通过 `start_response()` 发送Header，最后返回Body。

整个 `application()` 函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，我们只负责在更高层次上考虑如何响应请求就可以了。

不过，等等，这个 `application()` 函数怎么调用？如果我们自己调用，两个参数 `environ` 和 `start_response` 我们没法提供，返回的 `bytes` 也没法发给浏览器。

所以 `application()` 函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器，我们可以挑选一个来用。但是现在，我们只想尽快测试一下我们编写的 `application()` 函数真的可以把HTML输出到浏览器，所以，要赶紧找一个最简单的WSGI服务器，把我们的Web应用程序跑起来。

好消息是Python内置了一个WSGI服务器，这个模块叫wsgiref，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

运行WSGI服务

我们先编写 `hello.py` ，实现Web应用程序的WSGI处理函数：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']
```

然后，再编写一个 `server.py` ，负责启动WSGI服务器，加载 `application()` 函数：

```
# server.py
# 从wsgiref模块导入:
from wsgiref.simple_server import make_server
# 导入我们自己编写的application函数:
from hello import application

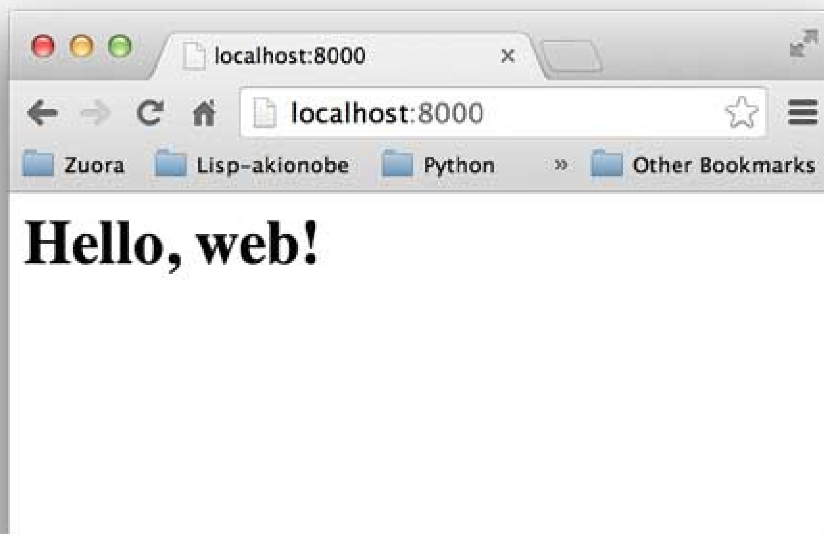
# 创建一个服务器，IP地址为空，端口是8000，处理函数是application:
httpd = make_server('', 8000, application)
print('Serving HTTP on port 8000...')
# 开始监听HTTP请求:
httpd.serve_forever()
```

确保以上两个文件在同一个目录下，然后在命令行输入 `python server.py` 来启动WSGI服务器：

```
michael — Python — 80×24
Michaels-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
```

注意：如果 `8000` 端口已被其他程序占用，启动将失败，请修改成其他端口。

启动成功后，打开浏览器，输入 `http://localhost:8000/` ，就可以看到结果了：



在命令行可以看到wsgiref打印的log信息：

```
michael — Python — 80×24
Michaels-MacBook-Pro-2:~ michael$ python server.py
Serving HTTP on port 8000...
127.0.0.1 - - [14/May/2014 11:01:11] "GET / HTTP/1.1" 200 20
127.0.0.1 - - [14/May/2014 11:01:11] "GET /favicon.ico HTTP/1.1" 200 20
```

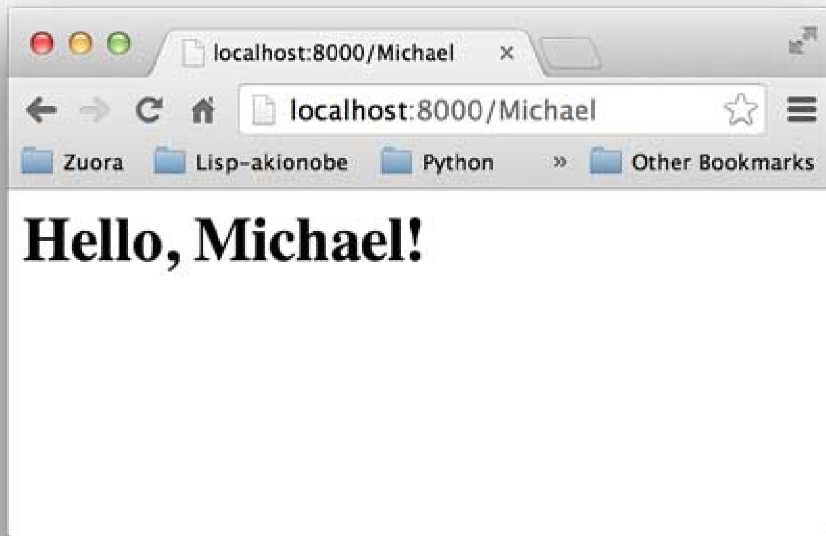
按 `Ctrl+C` 终止服务器。

如果你觉得这个Web应用太简单了，可以稍微改造一下，从 `environ` 里读取 `PATH_INFO` ，这样可以显示更加动态的内容：

```
# hello.py

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    body = '<h1>Hello, %s!</h1>' % (environ['PATH_INFO'][1:] or 'we
    return [body.encode('utf-8')]
```

你可以在地址栏输入用户名作为URL的一部分，将返回 `Hello, xxx!` ：



是不是有点Web App的感觉了？

小结

无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过 `environ` 获得，HTTP响应的输出都可以通过 `start_response()` 加上函数返回值作为Body。

复杂的Web应用程序，光靠一个WSGI函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架，进一步简化Web开发。

参考源码

hello.py

do_wsgi.py

使用Web框架

了解了WSGI框架，我们发现：其实一个Web App，就是写一个WSGI的处理函数，针对每个HTTP请求进行响应。

但是如何处理HTTP请求不是问题，问题是如何处理100个不同的URL。

每一个URL可以对应GET和POST请求，当然还有PUT、DELETE等请求，但是我们通常只考虑最常见的GET和POST请求。

一个最简单的想法是从 `environ` 变量里取出HTTP请求的信息，然后逐个判断：

```
def application(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    if method=='GET' and path=='/':
        return handle_home(environ, start_response)
    if method=='POST' and path='/signin':
        return handle_signin(environ, start_response)
    ...
```

只是这么写下去代码是肯定没法维护了。

代码这么写没法维护的原因是因为WSGI提供的接口虽然比HTTP接口高级了不少，但和Web App的处理逻辑比，还是比较低级，我们需要在WSGI接口之上能进一步抽象，让我们专注于用一个函数处理一个URL，至于URL到函数的映射，就交给Web框架来做。

由于用Python开发一个Web框架十分容易，所以Python有上百个开源的Web框架。这里我们先不讨论各种Web框架的优缺点，直接选择一个比较流行的Web框架——Flask来使用。

用Flask编写Web App比WSGI接口简单（这不是废话么，要是比WSGI还复杂，用框架干嘛？），我们先用 `pip` 安装Flask：

```
$ pip install flask
```

然后写一个`app.py`，处理3个URL，分别是：

- `GET /`：首页，返回`Home`；
- `GET /signin`：登录页，显示登录表单；
- `POST /signin`：处理登录表单，显示登录结果。

注意噢，同一个URL`/signin`分别有GET和POST两种请求，映射到两个处理函数中。

Flask通过Python的装饰器在内部自动地把URL和函数给关联起来，所以，我们写出来的代码就像这样：

```
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return '<h1>Home</h1>'

@app.route('/signin', methods=['GET'])
def signin_form():
    return '''<form action="/signin" method="post">
              <p><input name="username"></p>
              <p><input name="password" type="password"></p>
              <p><button type="submit">Sign In</button></p>
              </form>'''

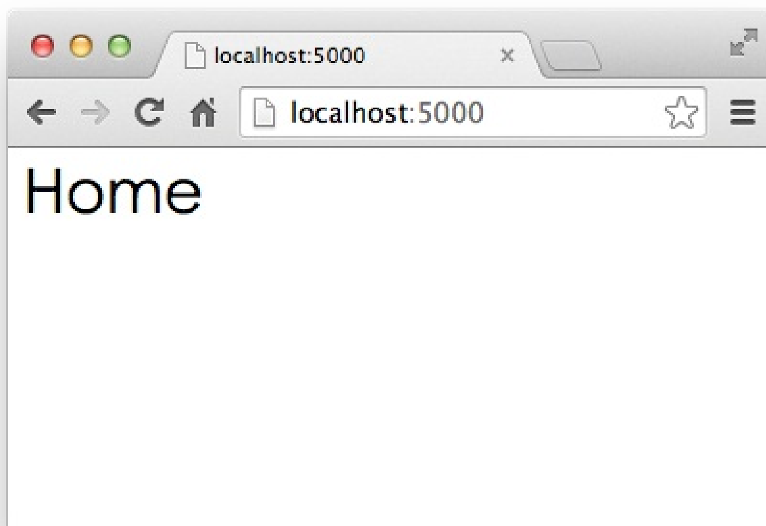
@app.route('/signin', methods=['POST'])
def signin():
    # 需要从request对象读取表单内容：
    if request.form['username']=='admin' and request.form['passwor
        return '<h3>Hello, admin!</h3>'
    return '<h3>Bad username or password.</h3>'

if __name__ == '__main__':
    app.run()
```

运行 `python app.py` ，Flask自带的Server在端口 `5000` 上监听：

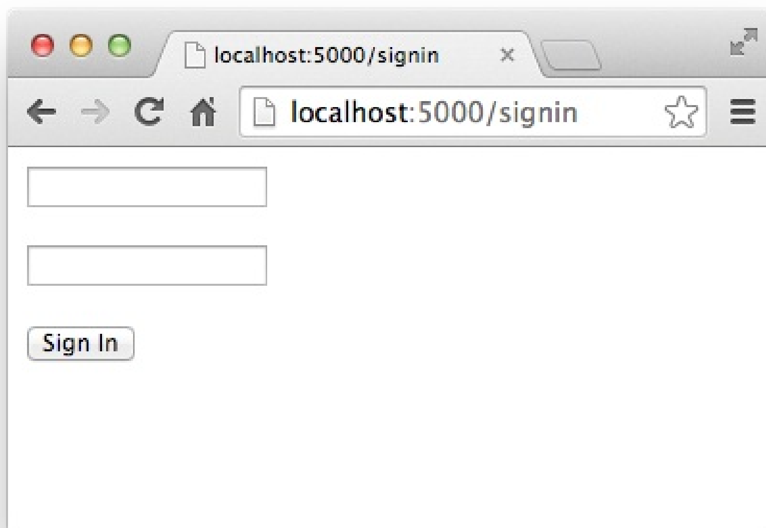
```
$ python app.py
 * Running on http://127.0.0.1:5000/
```

打开浏览器，输入首页地址 `http://localhost:5000/` ：

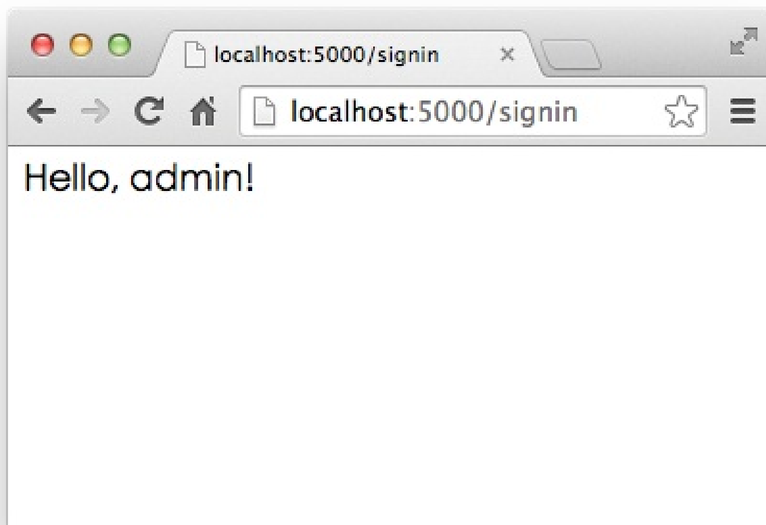


首页显示正确！

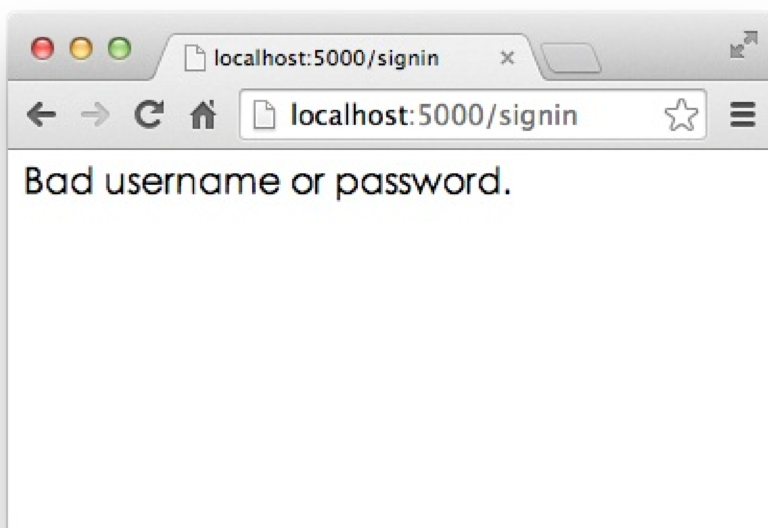
再在浏览器地址栏输入 `http://localhost:5000/signin` ，会显示登录表单：



输入预设的用户名 `admin` 和口令 `password` ，登录成功：



输入其他错误的用户名和口令，登录失败：



实际的Web App应该拿到用户名和口令后，去数据库查询再比对，来判断用户是否能登录成功。

除了Flask，常见的Python Web框架还有：

- Django：全能型Web框架；
- web.py：一个小巧的Web框架；
- Bottle：和Flask类似的Web框架；
- Tornado：Facebook的开源异步Web框架。

当然了，因为开发Python的Web框架也不是什么难事，我们后面也会讲到开发Web框架的内容。

小结

有了Web框架，我们在编写Web应用时，注意力就从WSGI处理函数转移到URL+对应的处理函数，这样，编写Web App就更加简单了。

在编写URL处理函数时，除了配置URL外，从HTTP请求拿到用户数据也是非常重要的。Web框架都提供了自己的API来实现这些功能。Flask通过 `request.form['name']` 来获取表单的内容。

参考源码

do_flask.py

使用模板

Web框架把我们从WSGI中拯救出来了。现在，我们只需要不断地编写函数，带上URL，就可以继续Web App的开发了。

但是，Web App不仅仅是处理逻辑，展示给用户的页面也非常重要。在函数中返回一个包含HTML的字符串，简单的页面还可以，但是，想想新浪首页的6000多行的HTML，你确信能在Python的字符串中正确地写出来么？反正我是做不到。

俗话说得好，不懂前端的Python工程师不是好的产品经理。有Web开发经验的同学都明白，Web App最复杂的部分就在HTML页面。HTML不仅要正确，还要通过CSS美化，再加上复杂的JavaScript脚本来实现各种交互和动画效果。总之，生成HTML页面的难度很大。

由于在Python代码里拼字符串是不现实的，所以，模板技术出现了。

使用模板，我们需要预先准备一个HTML文档，这个HTML文档不是普通的HTML，而是嵌入了一些变量和指令，然后，根据我们传入的数据，替换后，得到最终的HTML，发送给用户：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

Python处理URL的函数就是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量 `{{ name }}` 的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个 `dict` ：

```
{ 'name': 'Michael' }
```

只是因为Python支持关键字参数，很多Web框架允许传入关键字参数，然后，在框架内部组装出一个 `dict` 作为Model。

现在，我们把上次直接输出字符串作为HTML的例子用高端大气上档次的MVC模式改写一下：

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return render_template('home.html')

@app.route('/signin', methods=['GET'])
def signin_form():
    return render_template('form.html')

@app.route('/signin', methods=['POST'])
def signin():
    username = request.form['username']
    password = request.form['password']
    if username=='admin' and password=='password':
        return render_template('signin-ok.html', username=username)
    return render_template('form.html', message='Bad username or pa

if __name__ == '__main__':
    app.run()
```

Flask通过`render_template()`函数来实现模板的渲染。和Web框架类似，Python的模板也有很多种。Flask默认支持的模板是jinja2，所以我们先直接安装jinja2：

```
$ pip install jinja2
```

然后，开始编写jinja2模板：

home.html

用来显示首页的模板：

```
<html>
<head>
  <title>Home</title>
</head>
<body>
  <h1 style="font-style:italic">Home</h1>
</body>
</html>
```

form.html

用来显示登录表单的模板：

```
<html>
<head>
  <title>Please Sign In</title>
</head>
<body>
  {% if message %}
  <p style="color:red">{{ message }}</p>
  {% endif %}
  <form action="/signin" method="post">
    <legend>Please sign in:</legend>
    <p><input name="username" placeholder="Username" value="{{ use
    <p><input name="password" placeholder="Password" type="passwor
    <p><button type="submit">Sign In</button></p>
  </form>
</body>
</html>
```

signin-ok.html

登录成功的模板：

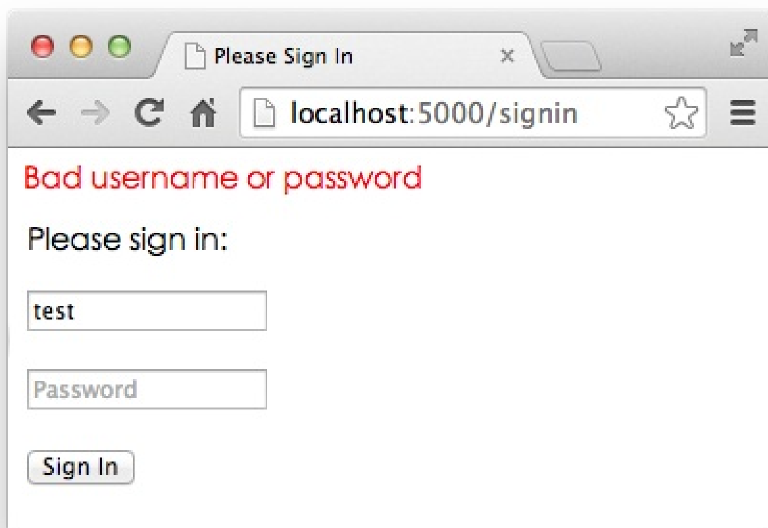
```
<html>
<head>
  <title>Welcome, {{ username }}</title>
</head>
<body>
  <p>Welcome, {{ username }}!</p>
</body>
</html>
```

登录失败的模板呢？我们在 `form.html` 中加了一点条件判断，把 `form.html` 重用为登录失败的模板。

最后，一定要把模板放到正确的 `templates` 目录下， `templates` 和 `app.py` 在同级目录下：



启动 `python app.py` ，看看使用模板的页面效果：



通过MVC，我们在Python代码中处理M：Model和C：Controller，而V：View是通过模板处理的，这样，我们就成功地把Python代码和HTML代码最大限度地分离了。

使用模板的另一大好处是，模板改起来很方便，而且，改完保存后，刷新浏览器就能看到最新的效果，这对于调试HTML、CSS和JavaScript的前端工程师来说实在是太重要了。

在Jinja2模板中，我们用 `{{ name }}` 表示一个需要替换的变量。很多时候，还需要循环、条件判断等指令语句，在Jinja2中，用 `{% ... %}` 表示指令。

比如循环输出页码：

```
{% for i in page_list %}
    <a href="/page/{{ i }}">{{ i }}</a>
{% endfor %}
```

如果 `page_list` 是一个list： `[1, 2, 3, 4, 5]` ，上面的模板将输出5个超链接。

除了Jinja2，常见的模板还有：

- Mako：用 `<% ... %>` 和 `${xxx}` 的一个模板；
- Cheetah：也是用 `<% ... %>` 和 `${xxx}` 的一个模板；
- Django：Django是一站式框架，内置一个用 `{% ... %}` 和 `{{ xxx }}` 的模板。

小结

有了MVC，我们就分离了Python代码和HTML代码。HTML代码全部放到模板里，写起来更有效率。

源码参考

app.py

异步IO

在IO编程一节中，我们已经知道，CPU的速度远远快于磁盘、网络等IO。在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。

在IO操作的过程中，当前线程被挂起，而其他需要CPU执行的代码就无法被当前线程执行了。

因为一个IO操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到IO导致线程被挂起，其他用户的线程不受影响。

多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。

由于我们要解决的问题是CPU高速执行能力和IO设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。

另一种解决IO问题的方法是异步IO。当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

可以想象如果按普通顺序写出的代码实际上是没法完成异步IO的：

```
do_some_code()
f = open('/path/to/file', 'r')
r = f.read() # <== 线程停在此处等待IO操作结果
# IO操作完成后线程才能继续执行:
do_some_code(r)
```

所以，同步IO模型的代码是无法实现异步IO模型的。

异步IO模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

消息模型其实早在应用在桌面应用程序中了。一个GUI程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到GUI程序的消息队列中，然后由GUI程序的主线程处理。

由于GUI线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个GUI程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。

消息模型是如何解决同步IO必须等待IO操作这一问题的呢？当遇到IO操作时，代码只负责发出IO请求，不等待IO结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当IO操作完成后，将收到一条“IO完成”的消息，处理该消息时就可以直接获取IO操作结果。

在“发出IO请求”到收到“IO完成”的这段时间里，同步IO模型下，主线程只能挂起，但异步IO模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步IO模型下，一个线程就可以同时处理多个IO请求，并且没有切换线程的操作。对于大多数IO密集型的应用程序，使用异步IO将大大提升系统的多任务处理能力。

协程

在学习异步IO模型前，我们先来了解协程。

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

```
def A():
    print('1')
    print('2')
    print('3')

def B():
    print('x')
    print('y')
    print('z')
```

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

```
1
2
x
y
3
z
```

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。

看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过 `for` 循环来迭代，还可以不断调用 `next()` 函数获取由 `yield` 语句返回的下一个值。

但是Python的 `yield` 不但可以返回一个值，它还可以接收调用者发出的参数。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过 `yield` 跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

c = consumer()
produce(c)
```

执行结果：

```
[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK
```

注意到 `consumer` 函数是一个 `generator` ，把一个 `consumer` 传入 `produce` 后：

1. 首先调用 `c.send(None)` 启动生成器；
2. 然后，一旦生产了东西，通过 `c.send(n)` 切换到 `consumer` 执行；
3. `consumer` 通过 `yield` 拿到消息，处理，又通过 `yield` 把结果传回；
4. `produce` 拿到 `consumer` 处理的结果，继续生产下一条消息；
5. `produce` 决定不生产了，通过 `c.close()` 关闭 `consumer` ，整个过程结束。

整个流程无锁，由一个线程执行， `produce` 和 `consumer` 协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：

“子程序就是协程的一种特例。”

参考源码

coroutine.py

asyncio

`asyncio` 是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

`asyncio` 的编程模型就是一个消息循环。我们从 `asyncio` 模块中直接获取一个 `EventLoop` 的引用，然后把需要执行的协程扔到 `EventLoop` 中执行，就实现了异步IO。

用 `asyncio` 实现 `Hello world` 代码如下：

```
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

`@asyncio.coroutine` 把一个generator标记为coroutine类型，然后，我们就把这个 `coroutine` 扔到 `EventLoop` 中执行。

`hello()` 会首先打印出 `Hello world!` ，然后， `yield from` 语法可以让我们方便地调用另一个 `generator` 。由于 `asyncio.sleep()` 也是一个 `coroutine` ，所以线程不会等待 `asyncio.sleep()` ，而是直接中断并执行下一个消息循环。当 `asyncio.sleep()` 返回时，线程就可以从 `yield from` 拿到返回值（此处是 `None` ），然后接着执行下一行语句。

把 `asyncio.sleep(1)` 看成是一个耗时1秒的IO操作，在此期间，主线程并未等待，而是去执行 `EventLoop` 中其他可以执行的 `coroutine` 了，因此可以实现并发执行。

我们用Task封装两个 `coroutine` 试试：

```
import threading
import asyncio

@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

观察执行过程：

```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
(暂停约1秒)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
```

由打印的当前线程名称可以看出，两个 `coroutine` 是由同一个线程并发执行的。

如果把 `asyncio.sleep()` 换成真正的IO操作，则多个 `coroutine` 就可以由一个线程并发执行。

我们用 `asyncio` 的异步网络连接来获取sina、sohu和163的网站首页：

```
import asyncio

@asyncio.coroutine
def wget(host):
    print('wget %s...' % host)
    connect = asyncio.open_connection(host, 80)
    reader, writer = yield from connect
    header = 'GET / HTTP/1.0\r\nHost: %s\r\n\r\n' % host
    writer.write(header.encode('utf-8'))
    yield from writer.drain()
    while True:
        line = yield from reader.readline()
        if line == b'\r\n':
            break
        print('%s header > %s' % (host, line.decode('utf-8').rstrip
    # Ignore the body, close the socket
    writer.close()

loop = asyncio.get_event_loop()
tasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com',
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

执行结果如下：

```
wget www.sohu.com...
wget www.sina.com.cn...
wget www.163.com...
(等待一段时间)
(打印出sohu的header)
www.sohu.com header > HTTP/1.1 200 OK
www.sohu.com header > Content-Type: text/html
...
(打印出sina的header)
www.sina.com.cn header > HTTP/1.1 200 OK
www.sina.com.cn header > Date: Wed, 20 May 2015 04:56:33 GMT
...
(打印出163的header)
www.163.com header > HTTP/1.0 302 Moved Temporarily
www.163.com header > Server: Cdn Cache Server V2.0
...
```

可见3个连接由一个线程通过 `coroutine` 并发完成。

小结

`asyncio` 提供了完善的异步IO支持；

异步操作需要在 `coroutine` 中通过 `yield from` 完成；

多个 `coroutine` 可以封装成一组Task然后并发执行。

参考源码

async_hello.py

async_wget.py

async/await

用 `asyncio` 提供的 `@asyncio.coroutine` 可以把一个generator标记为coroutine类型，然后在coroutine内部用 `yield from` 调用另一个coroutine实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法 `async` 和 `await` ，可以让coroutine的代码更简洁易读。

请注意， `async` 和 `await` 是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把 `@asyncio.coroutine` 替换为 `async` ；
2. 把 `yield from` 替换为 `await` 。

让我们对比一下上一节的代码：

```
@asyncio.coroutine
def hello():
    print("Hello world!")
    r = yield from asyncio.sleep(1)
    print("Hello again!")
```

用新语法重新编写如下：

```
async def hello():
    print("Hello world!")
    r = await asyncio.sleep(1)
    print("Hello again!")
```

剩下的代码保持不变。

小结

Python从3.5版本开始为 `asyncio` 提供了 `async` 和 `await` 的新语法；

注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

练习

将上一节的异步获取sina、sohu和163的网站首页源码用新语法重写并运行。

参考源码

async_hello2.py

async_wget2.py

aiohttp

`asyncio`可以实现单线程并发IO操作。如果仅用在客户端，发挥的威力不大。如果把`asyncio`用在服务器端，例如Web服务器，由于HTTP连接就是IO操作，因此可以用单线程+`coroutine`实现多用户的高并发支持。

`asyncio`实现了TCP、UDP、SSL等协议，`aiohttp`则是基于`asyncio`实现的HTTP框架。

我们先安装`aiohttp`：

```
pip install aiohttp
```

然后编写一个HTTP服务器，分别处理以下URL：

- `/` - 首页返回`b'<h1>Index</h1>'`；
- `/hello/{name}` - 根据URL参数返回文本`hello, %s!`。

代码如下：

```
import asyncio

from aiohttp import web

async def index(request):
    await asyncio.sleep(0.5)
    return web.Response(body=b'<h1>Index</h1>')

async def hello(request):
    await asyncio.sleep(0.5)
    text = '<h1>hello, %s!</h1>' % request.match_info['name']
    return web.Response(body=text.encode('utf-8'))

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    app.router.add_route('GET', '/hello/{name}', hello)
    srv = await loop.create_server(app.make_handler(), '127.0.0.1',
    print('Server started at http://127.0.0.1:8000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

注意 `aiohttp` 的初始化函数 `init()` 也是一个 `coroutine` ， `loop.create_server()` 则利用 `asyncio` 创建TCP服务。

参考源码

aio_web.py

实战

看完了教程，是不是有这么一种感觉：看的时候觉得很简单，照着教程敲代码也没啥大问题。

于是准备开始独立写代码，就发现不知道从哪开始下手了。

这种情况是完全正常的。好比学写作文，学的时候觉得简单，写的时候就无从下笔了。

虽然这个教程是面向小白的零基础Python教程，但是我们的目标不是学到60分，而是学到90分。

所以，用Python写一个真正的Web App吧！

目标

我们设定的实战目标是一个Blog网站，包含日志、用户和评论3大部分。

很多童鞋会想，这是不是太简单了？

比如webpy.org上就提供了一个Blog的例子，目测也就100行代码。

但是，这样的页面：

Hello, world

My first web app...

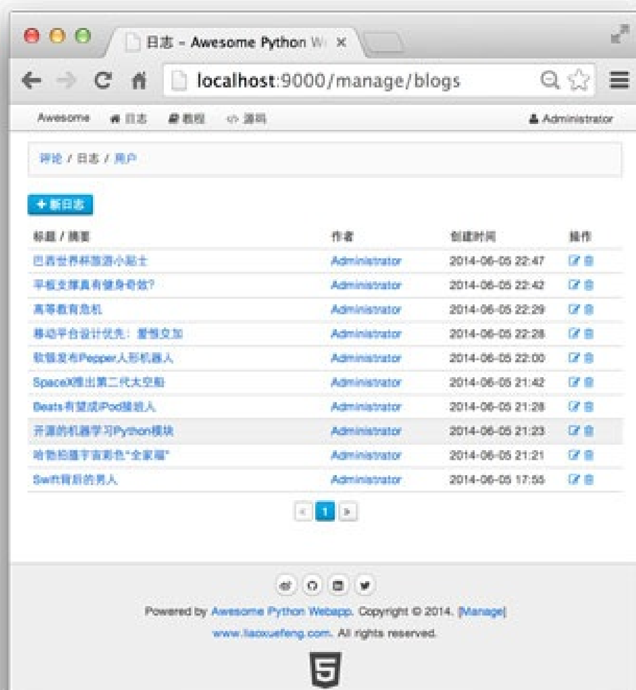
你拿得出手么？

我们要写出用户真正看得上眼的页面，首页长得像这样：

评论区：



还有极其强大的后台管理页面：



是不是一下子变得高端大气上档次了？

项目名称

必须是高端大气上档次的名称，命名为 `awesome-python3-webapp` 。

项目计划

项目计划开发周期为16天。每天，你需要完成教程中的内容。如果你觉得编写代码难度实在太大，可以参考一下当天在GitHub上的代码。

第N天的代码在 `https://github.com/michaelliao/awesome-python3-webapp/tree/day-N` 上。比如第1天就是：

https://github.com/michaelliao/awesome-python3-webapp/tree/day-01

以此类推。

要预览 `awesome-python3-webapp` 的最终页面效果，请猛击：

awesome.liaoxuefeng.com

Day 1 - 搭建开发环境

搭建开发环境

首先，确认系统安装的Python版本是3.4.x：

```
$ python3 --version
Python 3.4.3
```

然后，用 `pip` 安装开发Web App需要的第三方库：

异步框架aiohttp：

```
$pip3 install aiohttp
```

前端模板引擎jinja2：

```
$ pip3 install jinja2
```

MySQL 5.x数据库，从官方网站下载并安装，安装完毕后，请务必牢记root口令。为避免遗忘口令，建议直接把root口令设置为 `password` ；

MySQL的Python异步驱动程序aiomysql：

```
$ pip3 install aiomysql
```

项目结构

选择一个工作目录，然后，我们建立如下的目录结构：

```
awesome-python3-webapp/  <-- 根目录
|
+- backup/               <-- 备份目录
|
+- conf/                 <-- 配置文件
|
+- dist/                 <-- 打包目录
|
+- www/                  <-- Web目录，存放.py文件
|  |
|  +- static/            <-- 存放静态文件
|  |
|  +- templates/         <-- 存放模板文件
|
+- ios/                  <-- 存放iOS App工程
|
+- LICENSE               <-- 代码LICENSE
```

创建好项目的目录结构后，建议同时建立git仓库并同步至GitHub，保证代码修改的安全。

要了解git和GitHub的用法，请移步Git教程。

开发工具

自备，推荐用Sublime Text，请参考使用文本编辑器。

参考源码

day-01

Day 2 - 编写Web App骨架

由于我们的Web App建立在asyncio的基础上，因此用aiohttp写一个基本的 `app.py` ：

```
import logging; logging.basicConfig(level=logging.INFO)

import asyncio, os, json, time
from datetime import datetime

from aiohttp import web

def index(request):
    return web.Response(body=b'<h1>Awesome</h1>')

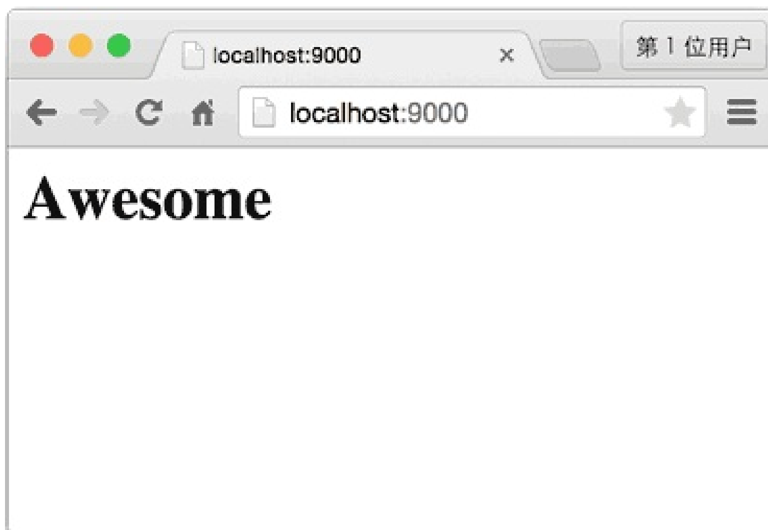
@asyncio.coroutine
def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    srv = yield from loop.create_server(app.make_handler(), '127.0.
    logging.info('server started at http://127.0.0.1:9000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

运行 `python app.py` ，Web App将在 `9000` 端口监听HTTP请求，并且对首页 `/` 进行响应：

```
$ python3 app.py
INFO:root:server started at http://127.0.0.1:9000...
```

这里我们简单地返回一个 `Awesome` 字符串，在浏览器中可以看到效果：



这说明我们的Web App骨架已经搭好了，可以进一步往里面添加更多的东西。

参考源码

day-02

Day 3 - 编写ORM

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在awesome-python3-webapp中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

所以，我们要首先把常用的SELECT、INSERT、UPDATE和DELETE操作用函数封装起来。

由于Web框架使用了基于asyncio的aiohttp，这是基于协程的异步模型。在协程中，不能调用普通的同步IO操作，因为所有用户都是由一个线程服务的，协程的执行速度必须非常快，才能处理大量用户的请求。而耗时的IO操作不能在协程中以同步的方式调用，否则，等待一个IO操作时，系统无法响应任何其他用户。

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步，“开弓没有回头箭”。

幸运的是 `aiomysql` 为MySQL数据库提供了异步IO的驱动。

创建连接池

我们需要创建一个全局的连接池，每个HTTP请求都可以从连接池中直接获取数据库连接。使用连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。

连接池由全局变量 `__pool` 存储，缺省情况下将编码设置为 `utf8` ，自动提交事务：

```
@asyncio.coroutine
def create_pool(loop, **kw):
    logging.info('create database connection pool...')
    global __pool
    __pool = yield from aiomysql.create_pool(
        host=kw.get('host', 'localhost'),
        port=kw.get('port', 3306),
        user=kw['user'],
        password=kw['password'],
        db=kw['db'],
        charset=kw.get('charset', 'utf8'),
        autocommit=kw.get('autocommit', True),
        maxsize=kw.get('maxsize', 10),
        minsize=kw.get('minsize', 1),
        loop=loop
    )
```

Select

要执行SELECT语句，我们用 `select` 函数执行，需要传入SQL语句和SQL参数：

```
@asyncio.coroutine
def select(sql, args, size=None):
    log(sql, args)
    global __pool
    with (yield from __pool) as conn:
        cur = yield from conn.cursor(aiomysql.DictCursor)
        yield from cur.execute(sql.replace('?', '%s'), args or ())
        if size:
            rs = yield from cur.fetchmany(size)
        else:
            rs = yield from cur.fetchall()
        yield from cur.close()
        logging.info('rows returned: %s' % len(rs))
        return rs
```

SQL语句的占位符是 `?` ，而MySQL的占位符是 `%s` ， `select()` 函数在内部自动替换。注意要始终坚持使用带参数的SQL，而不是自己拼接SQL字符串，这样可以防止SQL注入攻击。

注意到 `yield from` 将调用一个子协程（也就是在一个协程中调用另一个协程）并直接获得子协程的返回结果。

如果传入 `size` 参数，就通过 `fetchmany()` 获取最多指定数量的记录，否则，通过 `fetchall()` 获取所有记录。

Insert, Update, Delete

要执行INSERT、UPDATE、DELETE语句，可以定义一个通用的 `execute()` 函数，因为这3种SQL的执行都需要相同的参数，以及返回一个整数表示影响的行数：

```
@asyncio.coroutine
def execute(sql, args):
    log(sql)
    with (yield from __pool) as conn:
        try:
            cur = yield from conn.cursor()
            yield from cur.execute(sql.replace('?', '%s'), args)
            affected = cur.rowcount
            yield from cur.close()
        except BaseException as e:
            raise
        return affected
```

`execute()` 函数和 `select()` 函数所不同的是，cursor对象不返回结果集，而是通过 `rowcount` 返回结果数。

ORM

有了基本的 `select()` 和 `execute()` 函数，我们就可以开始编写一个简单的ORM了。

设计ORM需要从上层调用者角度来设计。

我们先考虑如何定义一个 `User` 对象，然后把数据库表 `users` 和它关联起来。

```
from orm import Model, StringField, IntegerField

class User(Model):
    __table__ = 'users'

    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在 `User` 类中的 `__table__` 、 `id` 和 `name` 是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述 `User` 对象和表的映射关系，而实例属性必须通过 `__init__()` 方法去初始化，所以两者互不干扰：

```
# 创建实例:
user = User(id=123, name='Michael')
# 存入数据库:
user.insert()
# 查询所有User对象:
users = User.findAll()
```

定义Model

首先要定义的是所有ORM映射的基类 `Model` ：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute

    def __setattr__(self, key, value):
        self[key] = value

    def getValue(self, key):
        return getattr(self, key, None)

    def getValueOrDefault(self, key):
        value = getattr(self, key, None)
        if value is None:
            field = self.__mappings__[key]
            if field.default is not None:
                value = field.default() if callable(field.default)
                logging.debug('using default value for %s: %s' % (k
                setattr(self, key, value)
        return value
```

`Model` 从 `dict` 继承，所以具备所有 `dict` 的功能，同时又实现了特殊方法 `__getattr__()` 和 `__setattr__()`，因此又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

以及 `Field` 和各种 `Field` 子类：

```
class Field(object):

    def __init__(self, name, column_type, primary_key, default):
        self.name = name
        self.column_type = column_type
        self.primary_key = primary_key
        self.default = default

    def __str__(self):
        return '<%s, %s:%s>' % (self.__class__.__name__, self.colur
```

映射 `varchar` 的 `StringField` ：

```
class StringField(Field):

    def __init__(self, name=None, primary_key=False, default=None,
        super().__init__(name, ddl, primary_key, default)
```

注意到 `Model` 只是一个基类，如何将具体的子类如 `User` 的映射信息读取出来呢？答案就是通过metaclass： `ModelMetaclass` ：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        # 排除Model类本身:
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        # 获取table名称:
        tableName = attrs.get('__table__', None) or name
        logging.info('found model: %s (table: %s)' % (name, tableNa
        # 获取所有的Field和主键名:
        mappings = dict()
        fields = []
        primaryKey = None
        for k, v in attrs.items():
            if isinstance(v, Field):
                logging.info('  found mapping: %s ==> %s' % (k, v))
```

```
                mappings[k] = v
                if v.primary_key:
                    # 找到主键:
                    if primaryKey:
                        raise RuntimeError('Duplicate primary key f
                    primaryKey = k
                else:
                    fields.append(k)
        if not primaryKey:
            raise RuntimeError('Primary key not found.')
        for k in mappings.keys():
            attrs.pop(k)
        escaped_fields = list(map(lambda f: '`%s`' % f, fields))
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = tableName
        attrs['__primary_key__'] = primaryKey # 主键属性名
        attrs['__fields__'] = fields # 除主键外的属性名
        # 构造默认的SELECT, INSERT, UPDATE和DELETE语句:
        attrs['__select__'] = 'select `%s`, %s from `%s`' % (primar
        attrs['__insert__'] = 'insert into `%s` (%s, `%s`) values (
        attrs['__update__'] = 'update `%s` set %s where `%s`=?' % (
        attrs['__delete__'] = 'delete from `%s` where `%s`=?' % (ta
        return type.__new__(cls, name, bases, attrs)
```

这样，任何继承自Model的类（比如User），会自动通过ModelMetaclass扫描映射关系，并存储到自身的类属性如 `__table__` 、 `__mappings__` 中。

然后，我们往Model类添加class方法，就可以让所有子类调用class方法：

```
class Model(dict):

    ...

    @classmethod
    @asyncio.coroutine
    def find(cls, pk):
        ' find object by primary key. '
        rs = yield from select('%s where `%s`=?' % (cls.__select__,
        if len(rs) == 0:
            return None
        return cls(**rs[0])
```

User类现在就可以通过类方法实现主键查找：

```
user = yield from User.find('123')
```

往Model类添加实例方法，就可以让所有子类调用实例方法：

```
class Model(dict):

    ...

    @asyncio.coroutine
    def save(self):
        args = list(map(self.getValueOrDefault, self.__fields__))
        args.append(self.getValueOrDefault(self.__primary_key__))
        rows = yield from execute(self.__insert__, args)
        if rows != 1:
            logging.warn('failed to insert record: affected rows: %
```

这样，就可以把一个User实例存入数据库：

```
user = User(id=123, name='Michael')
yield from user.save()
```

最后一步是完善ORM，对于查找，我们可以实现以下方法：

- findAll() - 根据WHERE条件查找；
- findNumber() - 根据WHERE条件查找，但返回的是整数，适用于 `select count(*)` 类型的SQL。

以及 `update()` 和 `remove()` 方法。

所有这些方法都必须用 `@asyncio.coroutine` 装饰，变成一个协程。

调用时需要特别注意：

```
user.save()
```

没有任何效果，因为调用 `save()` 仅仅是创建了一个协程，并没有执行它。一定要用：

```
yield from user.save()
```

才真正执行了INSERT操作。

最后看看我们实现的ORM模块一共多少行代码？累计不到300多行。用Python写一个ORM是不是很容易呢？

参考源码

day-03

Day 4 - 编写Model

有了ORM，我们就可以把Web App需要的3个表用 `Model` 表示出来：

```
import time, uuid

from orm import Model, StringField, BooleanField, FloatField, TextF

def next_id():
    return '%015d%s000' % (int(time.time() * 1000), uuid.uuid4().he

class User(Model):
    __table__ = 'users'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
    email = StringField(ddl='varchar(50)')
    passwd = StringField(ddl='varchar(50)')
    admin = BooleanField()
    name = StringField(ddl='varchar(50)')
    image = StringField(ddl='varchar(500)')
    created_at = FloatField(default=time.time)

class Blog(Model):
    __table__ = 'blogs'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(default=time.time)

class Comment(Model):
    __table__ = 'comments'

    id = StringField(primary_key=True, default=next_id, ddl='varcha
```

```
    blog_id = StringField(ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(default=time.time)
```

在编写ORM时，给一个Field增加一个 `default` 参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用 `save()` 时自动计算。

例如，主键 `id` 的缺省值是函数 `next_id` ，创建时间 `created_at` 的缺省值是函数 `time.time` ，可以自动设置当前日期和时间。

日期和时间用 `float` 类型存储在数据库中，而不是 `datetime` 类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个 `float` 到 `str` 的转换，也非常容易。

初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

```
-- schema.sql

drop database if exists awesome;

create database awesome;

use awesome;

grant select, insert, update, delete on awesome.* to 'www-data'@'lo

create table users (
    `id` varchar(50) not null,
    `email` varchar(50) not null,
    `passwd` varchar(50) not null,
    `admin` bool not null,
    `name` varchar(50) not null,
```

```
    `image` varchar(500) not null,
    `created_at` real not null,
    unique key `idx_email` (`email`),
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table blogs (
    `id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `name` varchar(50) not null,
    `summary` varchar(200) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;

create table comments (
    `id` varchar(50) not null,
    `blog_id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
) engine=innodb default charset=utf8;
```

如果表的数量很多，可以从 `Model` 对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

```
$ mysql -u root -p < schema.sql
```

我们就完成了数据库表的初始化。

编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于 `User` 对象，我们就可以做如下操作：

```
import orm
from models import User, Blog, Comment

def test():
    yield from orm.create_pool(user='www-data', password='www-data

    u = User(name='Test', email='test@example.com', passwd='1234567

    yield from u.save()

for x in test():
    pass
```

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

参考源码

day-04

Day 5 - 编写Web框架

在正式开始Web开发前，我们需要编写一个Web框架。

`aiohttp` 已经是一个Web框架了，为什么我们还需要自己封装一个？

原因是从使用者的角度来说， `aiohttp` 相对比较底层，编写一个URL的处理函数需要这么几步：

第一步，编写一个用 `@asyncio.coroutine` 装饰的函数：

```
@asyncio.coroutine
def handle_url_xxx(request):
    pass
```

第二步，传入的参数需要自己从 `request` 中获取：

```
url_param = request.match_info['key']
query_params = parse_qs(request.query_string)
```

最后，需要自己构造 `Response` 对象：

```
text = render('template', data)
return web.Response(text.encode('utf-8'))
```

这些重复的工作可以由框架完成。例如，处理带参数的URL `/blog/{id}` 可以这么写：

```
@get('/blog/{id}')
def get_blog(id):
    pass
```

处理 `query_string` 参数可以通过关键字参数 `**kw` 或者命名关键字参数接收：

```
@get('/api/comments')
def api_comments(*, page='1'):
    pass
```

对于函数的返回值，不一定是 `web.Response` 对象，可以是 `str` 、 `bytes` 或 `dict` 。

如果希望渲染模板，我们可以这么返回一个 `dict` ：

```
return {
    '__template__': 'index.html',
    'data': '...'
}
```

因此，Web框架的设计是完全从使用者出发，目的是让使用者编写尽可能少的代码。

编写简单的函数而非引入 `request` 和 `web.Response` 还有一个额外的好处，就是可以单独测试，否则，需要模拟一个 `request` 才能测试。

@get和@post

要把一个函数映射为一个URL处理函数，我们先定义 `@get()` ：

```
def get(path):
    '''
    Define decorator @get('/path')
    '''
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            return func(*args, **kw)
        wrapper.__method__ = 'GET'
        wrapper.__route__ = path
        return wrapper
    return decorator
```

这样，一个函数通过 `@get()` 的装饰就附带了URL信息。

`@post` 与 `@get` 定义类似。

定义RequestHandler

URL处理函数不一定是一个 `coroutine` ，因此我们用 `RequestHandler()` 来封装一个URL处理函数。

`RequestHandler` 是一个类，由于定义了 `__call__()` 方法，因此可以将其实例视为函数。

`RequestHandler` 目的就是从URL函数中分析其需要接收的参数，从 `request` 中获取必要的参数，调用URL函数，然后把结果转换为 `web.Response` 对象，这样，就完全符合 `aiohttp` 框架的要求：

```
class RequestHandler(object):

    def __init__(self, app, fn):
        self._app = app
        self._func = fn
        ...

    @asyncio.coroutine
    def __call__(self, request):
        kw = ... 获取参数
        r = yield from self._func(**kw)
        return r
```

再编写一个 `add_route` 函数，用来注册一个URL处理函数：

```
def add_route(app, fn):
    method = getattr(fn, '__method__', None)
    path = getattr(fn, '__route__', None)
    if path is None or method is None:
        raise ValueError('@get or @post not defined in %s.' % str(f
    if not asyncio.iscoroutinefunction(fn) and not inspect.isgenera
        fn = asyncio.coroutine(fn)
    logging.info('add route %s %s => %s(%s)' % (method, path, fn.__
    app.router.add_route(method, path, RequestHandler(app, fn))
```

最后一步，把很多次`add_route()`注册的调用：

```
add_route(app, handles.index)
add_route(app, handles.blog)
add_route(app, handles.create_comment)
...
```

变成自动扫描：

```
# 自动把handler模块的所有符合条件的函数注册了:
add_routes(app, 'handlers')
```

`add_routes()`定义如下：

```
def add_routes(app, module_name):
    n = module_name.rfind('.')
    if n == (-1):
        mod = __import__(module_name, globals(), locals())
    else:
        name = module_name[n+1:]
        mod = getattr(__import__(module_name[:n], globals(), locals
    for attr in dir(mod):
        if attr.startswith('_'):
            continue
        fn = getattr(mod, attr)
        if callable(fn):
            method = getattr(fn, '__method__', None)
            path = getattr(fn, '__route__', None)
            if method and path:
                add_route(app, fn)
```

最后，在 `app.py` 中加入 `middleware` 、 `jinja2` 模板和自注册的支持：

```
app = web.Application(loop=loop, middlewares=[
    logger_factory, response_factory
])
init_jinja2(app, filters=dict(datetime=datetime_filter))
add_routes(app, 'handlers')
add_static(app)
```

middleware

`middleware` 是一种拦截器，一个URL在被某个函数处理前，可以经过一系列的 `middleware` 的处理。

一个 `middleware` 可以改变URL的输入、输出，甚至可以决定不继续处理而直接返回。middleware的用处就在于把通用的功能从每个URL处理函数中拿出来，集中放到一个地方。例如，一个记录URL日志的 `logger` 可以简单定义如下：

```
@asyncio.coroutine
def logger_factory(app, handler):
    @asyncio.coroutine
    def logger(request):
        # 记录日志:
        logging.info('Request: %s %s' % (request.method, request.pa
        # 继续处理请求:
        return (yield from handler(request))
    return logger
```

而 `response` 这个 `middleware` 把返回值转换为 `web.Response` 对象再返回，以保证满足 `aiohttp` 的要求：

```
@asyncio.coroutine
def response_factory(app, handler):
    @asyncio.coroutine
    def response(request):
        # 结果:
        r = yield from handler(request)
        if isinstance(r, web.StreamResponse):
            return r
        if isinstance(r, bytes):
            resp = web.Response(body=r)
            resp.content_type = 'application/octet-stream'
            return resp
        if isinstance(r, str):
            resp = web.Response(body=r.encode('utf-8'))
            resp.content_type = 'text/html;charset=utf-8'
            return resp
        if isinstance(r, dict):
            ...
```

有了这些基础设施，我们就可以专注地往 `handlers` 模块不断添加URL处理函数了，可以极大地提高开发效率。

参考源码

day-05

Day 6 - 编写配置文件

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的 `.properties` 或者 `.yaml` 等配置文件。

默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置文件命名为 `config_default.py` ：

```
# config_default.py

configs = {
    'db': {
        'host': '127.0.0.1',
        'port': 3306,
        'user': 'www-data',
        'password': 'www-data',
        'database': 'awesome'
    },
    'session': {
        'secret': 'AwEsOmE'
    }
}
```

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的host等信息，直接修改 `config_default.py` 不是一个好办法，更好的方法是编写一个 `config_override.py` ，用来覆盖某些默认设置：

```
# config_override.py

configs = {
    'db': {
        'host': '192.168.0.100'
    }
}
```

把 `config_default.py` 作为开发环境的标准配置，把 `config_override.py` 作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从 `config_override.py` 读取。为了简化读取配置文件，可以把所有配置读取到统一的 `config.py` 中：

```
# config.py
configs = config_default.configs

try:
    import config_override
    configs = merge(configs, config_override.configs)
except ImportError:
    pass
```

这样，我们就完成了App的配置。

参考源码

day-06

Day 7 - 编写MVC

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的 `@get` 和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

```
@get('/')
def index(request):
    users = yield from User.findAll()
    return {
        '__template__': 'test.html',
        'users': users
    }
```

`'__template__'` 指定的模板文件是 `test.html` ，其他参数是传递给模板的数据，所以我们在模板的根目录 `templates` 下创建 `test.html` ：

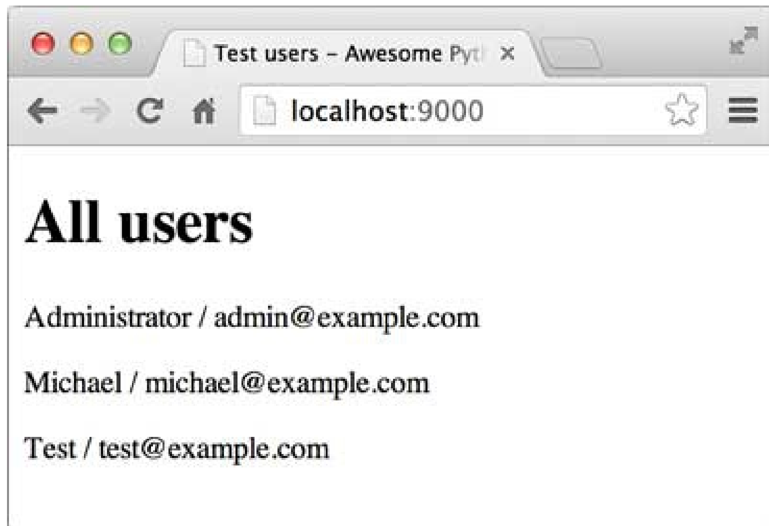
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test users - Awesome Python Webapp</title>
</head>
<body>
    <h1>All users</h1>
    {% for u in users %}
    <p>{{ u.name }} / {{ u.email }}</p>
    {% endfor %}
</body>
</html>
```

接下来，如果一切顺利，可以用命令行启动Web服务器：

```
$ python3 app.py
```

然后，在浏览器中访问 `http://localhost:9000/` 。

如果数据库的 `users` 表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给 `users` 表添加几条记录，然后再访问：



参考源码

day-07

Day 8 - 构建前端

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择uikit这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从uikit首页下载打包的资源文件。

所有的静态资源文件我们统一放到 `www/static` 目录下，并按照类别归类：

```
static/
+- css/
|  +- addons/
|  |  +- uikit.addons.min.css
|  |  +- uikit.almost-flat.addons.min.css
|  |  +- uikit.gradient.addons.min.css
|  +- awesome.css
|  +- uikit.almost-flat.addons.min.css
|  +- uikit.gradient.addons.min.css
|  +- uikit.min.css
+- fonts/
|  +- fontawesome-webfont.eot
|  +- fontawesome-webfont.ttf
|  +- fontawesome-webfont.woff
|  +- FontAwesome.otf
+- js/
   +- awesome.js
   +- html5.js
   +- jquery.min.js
   +- uikit.min.js
```

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

```
<html>
    <% include file="inc_header.html" %>
    <% include file="index_body.html" %>
    <% include file="inc_footer.html" %>
</html>
```

这样，相同的部分 `inc_header.html` 和 `inc_footer.html` 就可以共享。

但是include方法不利于页面整体结构的维护。jinjia2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

```
<!-- base.html -->
<html>
    <head>
        <title>{% block title%} 这里定义了一个名为title的block {% endb
    </head>
    <body>
        {% block content %} 这里定义了一个名为content的block {% endblo
    </body>
</html>
```

对于子模板 `a.html` ，只需要把父模板的 `title` 和 `content` 替换掉：

```
{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}
    <h1>Chapter A</h1>
    <p>blablabla...</p>
{% endblock %}
```

对于子模板 `b.html`，如法炮制：

```
{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}
    <h1>Chapter B</h1>
    <ul>
       <li>list 1</li>
       <li>list 2</li>
    </ul>
{% endblock %}
```

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板 `__base__.html` 的编写：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    {% block meta %}<!-- block meta  -->{% endblock %}
    <title>{% block title %} ? {% endblock %} - Awesome Python Weba
    <link rel="stylesheet" href="/static/css/uikit.min.css">
    <link rel="stylesheet" href="/static/css/uikit.gradient.min.css
    <link rel="stylesheet" href="/static/css/awesome.css" />
    <script src="/static/js/jquery.min.js"></script>
    <script src="/static/js/md5.js"></script>
```

```
    <script src="/static/js/uikit.min.js"></script>
    <script src="/static/js/awesome.js"></script>
    {% block beforehead %}<!-- before head  -->{% endblock %}
</head>
<body>
    <nav class="uk-navbar uk-navbar-attached uk-margin-bottom">
        <div class="uk-container uk-container-center">
            <a href="/" class="uk-navbar-brand">Awesome</a>
            <ul class="uk-navbar-nav">
                <li data-url="blogs"><a href="/"><i class="uk-icon-
                <li><a target="_blank" href="#"><i class="uk-icon-b
                <li><a target="_blank" href="#"><i class="uk-icon-c
            </ul>
            <div class="uk-navbar-flip">
                <ul class="uk-navbar-nav">
                {% if user %}
                    <li class="uk-parent" data-uk-dropdown>
                        <a href="#0"><i class="uk-icon-user"></i> <
                        <div class="uk-dropdown uk-dropdown-navbar"
                            <ul class="uk-nav uk-nav-navbar">
                                <li><a href="/signout"><i class="uk
                            </ul>
                        </div>
                    </li>
                {% else %}
                    <li><a href="/signin"><i class="uk-icon-sign-in
                    <li><a href="/register"><i class="uk-icon-edit"
                {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <div class="uk-container uk-container-center">
        <div class="uk-grid">
            <!-- content -->
            {% block content %}
            {% endblock %}
            <!-- // content -->
        </div>
```

```
    </div>

    <div class="uk-margin-large-top" style="background-color:#eee;
        <div class="uk-container uk-container-center uk-text-center
            <div class="uk-panel uk-margin-top uk-margin-bottom">
                <p>
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                    <a target="_blank" href="#" class="uk-icon-butt
                </p>
                <p>Powered by <a href="#">Awesome Python Webapp</a>
                <p><a href="http://www.liaoxuefeng.com/" target="_b
                <a target="_blank" href="#"><i class="uk-icon-html5
            </div>
        </div>
    </div>
</body>
</html>
```

`__base__.html` 定义的几个block作用如下：

用于子页面定义一些meta，例如rss feed：

```
{% block meta %} ... {% endblock %}
```

覆盖页面的标题：

```
{% block title %} ... {% endblock %}
```

子页面可以在 `<head>` 标签关闭前插入JavaScript代码：

```
{% block beforehead %} ... {% endblock %}
```

子页面的content布局和内容：

```
{% block content %}
    ...
{% endblock %}
```

我们把首页改造一下，从 `__base__.html` 继承一个 `blogs.html` ：

```
{% extends '__base__.html' %}

{% block title %}日志{% endblock %}

{% block content %}

    <div class="uk-width-medium-3-4">
        {% for blog in blogs %}
            <article class="uk-article">
                <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</
                <p class="uk-article-meta">发表于{{ blog.created_at
                <p>{{ blog.summary }}</p>
                <p><a href="/blog/{{ blog.id }}">继续阅读 <i class="
            </article>
            <hr class="uk-article-divider">
        {% endfor %}
    </div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
                <li><i class="uk-icon-thumbs-o-up"></i> <a target='
            </ul>
        </div>
    </div>

{% endblock %}
```

相应地，首页URL的处理函数更新如下：

```
@get('/')
def index(request):
    summary = 'Lorem ipsum dolor sit amet, consectetur adipisicing
    blogs = [
        Blog(id='1', name='Test Blog', summary=summary, created_at=
        Blog(id='2', name='Something New', summary=summary, createc
        Blog(id='3', name='Learn Swift', summary=summary, created_a
    ]
    return {
        '__template__': 'blogs.html',
        'blogs': blogs
    }
```

Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

```
<p class="uk-article-meta">发表于{{ blog.created_at }}</p>
```

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个 `datetime` 的filter，在模板里用法如下：

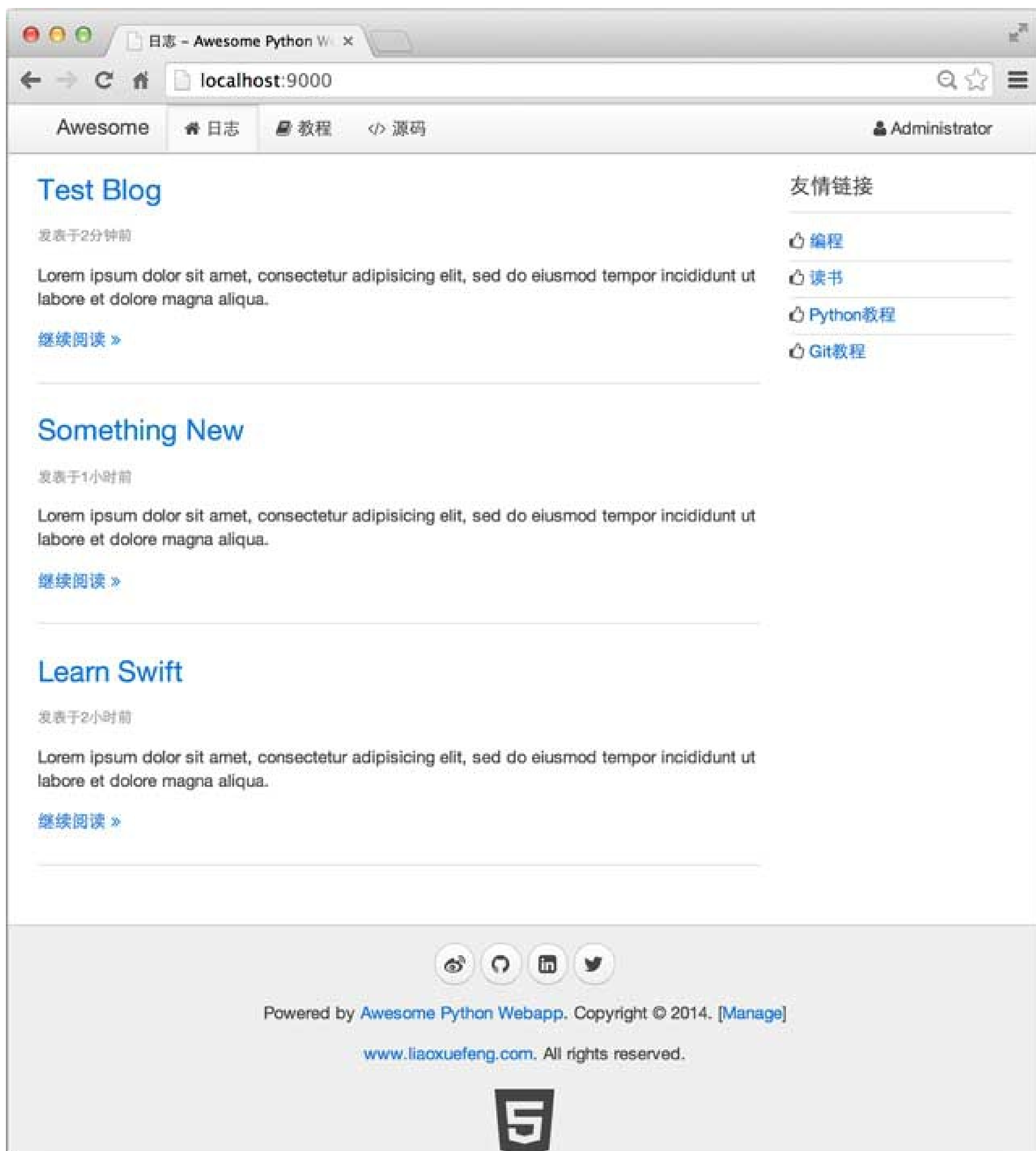
```
<p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
```

filter需要在初始化jinja2时设置。相关代码如下：

```
def datetime_filter(t):
    delta = int(time.time() - t)
    if delta < 60:
        return '1分钟前'
    if delta < 3600:
        return '%s分钟前' % (delta // 60)
    if delta < 86400:
        return '%s小时前' % (delta // 3600)
    if delta < 604800:
        return '%s天前' % (delta // 86400)
    dt = datetime.fromtimestamp(t)
    return '%s年%s月%s日' % (dt.year, dt.month, dt.day)

...
init_jinja2(app, filters=dict(datetime=datetime_filter))
...
```

现在，完善的首页显示如下：


日志 - Awesome Python W
localhost:9000
Awesome
日志
教程
源码
Administrator
Test Blog
发表于2分钟前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
Something New
发表于1小时前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
Learn Swift
发表于2小时前
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
继续阅读 »
友情链接
编程
读书
Python教程
Git教程
Powered by Awesome Python Webapp. Copyright © 2014. [Manage]
www.liaoxuefeng.com. All rights reserved.

参考源码

day-08

Day 9 - 编写API

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入 `http://localhost:9000/blog/123` ，就可以看到id为 `123` 的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取 `http://localhost:9000/api/blogs/123` ，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个 `@api` 来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

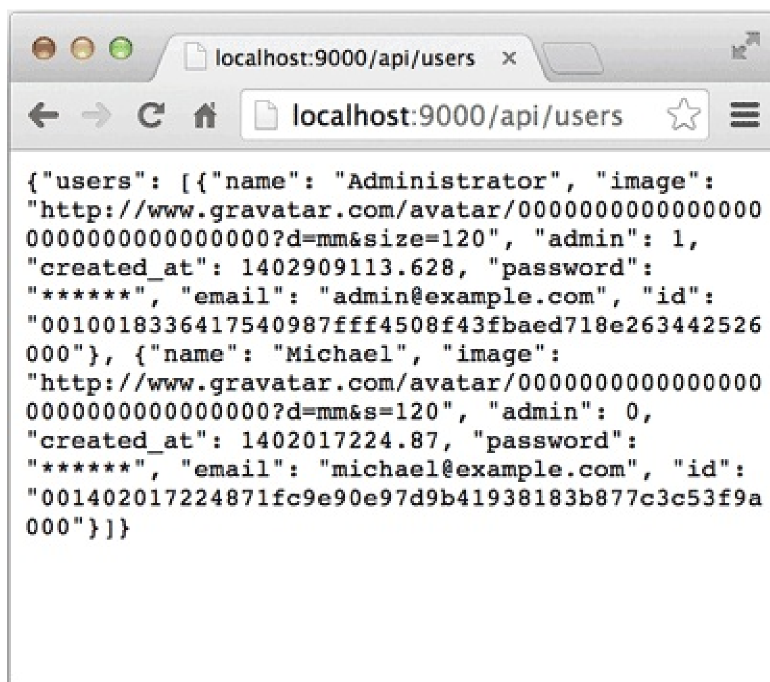
```
@get('/api/users')
def api_get_users(*, page='1'):
    page_index = get_page_index(page)
    num = yield from User.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, users=())
    users = yield from User.findAll(orderBy='created_at desc', limi
    for u in users:
        u.passwd = '******'
    return dict(page=p, users=users)
```

只要返回一个 `dict` ，后续的 `response` 这个 `middleware` 就可以把结果序列化为JSON并返回。

我们需要对Error进行处理，因此定义一个 `APIError` ，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为 `internalerror` 。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入 `http://localhost:9000/api/users` ，就可以看到返回的JSON：


localhost:9000/api/users
localhost:9000/api/users
{"users": [{"name": "Administrator", "image":
"http://www.gravatar.com/avatar/0000000000000000
0000000000000000?d=mm&size=120", "admin": 1,
"created_at": 1402909113.628, "password":
"******", "email": "admin@example.com", "id":
"0010018336417540987fff4508f43fbaed718e263442526
000"}, {"name": "Michael", "image":
"http://www.gravatar.com/avatar/0000000000000000
0000000000000000?d=mm&s=120", "admin": 0,
"created_at": 1402017224.87, "password":
"******", "email": "michael@example.com", "id":
"001402017224871fc9e90e97d9b41938183b877c3c53f9a
000"}]}

参考源码

day-09

Day 10 - 用户注册和登录

用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

```
_RE_EMAIL = re.compile(r'^[a-z0-9\.\-\_]+\@[a-z0-9\-\_]+(\.[a-z0-9\
_RE_SHA1 = re.compile(r'^[0-9a-f]{40}$')

@post('/api/users')
def api_register_user(*, email, name, passwd):
    if not name or not name.strip():
        raise APIValueError('name')
    if not email or not _RE_EMAIL.match(email):
        raise APIValueError('email')
    if not passwd or not _RE_SHA1.match(passwd):
        raise APIValueError('passwd')
    users = yield from User.findAll('email=?', [email])
    if len(users) > 0:
        raise APIError('register:failed', 'email', 'Email is alread
    uid = next_id()
    sha1_passwd = '%s:%s' % (uid, passwd)
    user = User(id=uid, name=name.strip(), email=email, passwd=hash
    yield from user.save()
    # make session cookie:
    r = web.Response()
    r.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=864
    user.passwd = '******'
    r.content_type = 'application/json'
    r.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
    return r
```

注意用户口令是客户端传递的经过SHA1计算后的40位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

```
{% extends '__base__.html' %}

{% block title %}注册{% endblock %}

{% block beforehead %}

<script>
function validateEmail(email) {
    var re = /^[a-z0-9\.\-\_]+\@[a-z0-9\-\_]+(\.[a-z0-9\-\_]+){1,4]
    return re.test(email.toLowerCase());
}
$(function () {
    var vm = new Vue({
        el: '#vm',
        data: {
            name: '',
            email: '',
            password1: '',
            password2: ''
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm');
                if (! this.name.trim()) {
                    return $form.showFormError('请输入名字');
                }
                if (! validateEmail(this.email.trim().toLowerCase()
                    return $form.showFormError('请输入正确的Email地址
                }
                if (this.password1.length < 6) {
                    return $form.showFormError('口令长度至少为6个字符'
                }
                if (this.password1 !== this.password2) {
                    return $form.showFormError('两次输入的口令不一致')
                }
                var email = this.email.trim().toLowerCase();
```

```
                $form.postJSON('/api/users', {
                    name: this.name.trim(),
                    email: email,
                    passwd: CryptoJS.SHA1(email + ':' + this.passwo
                }, function (err, r) {
                    if (err) {
                        return $form.showFormError(err);
                    }
                    return location.assign('/');
                });
            }
        }
    });
    $('#vm').show();
});
</script>

{% endblock %}

{% block content %}

    <div class="uk-width-2-3">
        <h1>欢迎注册！</h1>
        <form id="vm" v-on="submit: submit" class="uk-form uk-form-
            <div class="uk-alert uk-alert-danger uk-hidden"></div>
            <div class="uk-form-row">
                <label class="uk-form-label">名字:</label>
                <div class="uk-form-controls">
                    <input v-model="name" type="text" maxlength="50
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">电子邮件:</label>
                <div class="uk-form-controls">
                    <input v-model="email" type="text" maxlength="5
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">输入口令:</label>
                <div class="uk-form-controls">
```

```
                    <input v-model="password1" type="password" maxl
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">重复口令:</label>
                <div class="uk-form-controls">
                    <input v-model="password2" type="password" maxl
                </div>
            </div>
            <div class="uk-form-row">
                <button type="submit" class="uk-button uk-button-pr
            </div>
        </form>
    </div>

{% endblock %}
```

这样我们就把用户注册的功能完成了：

用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如SHA1），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

```
"用户id" + "过期时间" + SHA1("用户id" + "用户口令" + "过期时间" + "Secre
```

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

- 用户id
- 过期时间
- SHA1值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

```
SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

并与浏览器cookie中的MD5进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于SHA1是一种单向算法，即可以通过原始字符串计算出SHA1结果，但无法通过SHA1结果反推出原始字符串。

所以登录API可以实现如下：

```
@post('/api/authenticate')
def authenticate(*, email, passwd):
    if not email:
        raise APIValueError('email', 'Invalid email.')
    if not passwd:
        raise APIValueError('passwd', 'Invalid password.')
    users = yield from User.findAll('email=?', [email])
    if len(users) == 0:
        raise APIValueError('email', 'Email not exist.')
    user = users[0]
    # check passwd:
    sha1 = hashlib.sha1()
    sha1.update(user.id.encode('utf-8'))
    sha1.update(b':')
    sha1.update(passwd.encode('utf-8'))
    if user.passwd != sha1.hexdigest():
        raise APIValueError('passwd', 'Invalid password.')
    # authenticate ok, set cookie:
    r = web.Response()
    r.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=864
    user.passwd = '******'
    r.content_type = 'application/json'
    r.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
    return r

# 计算加密cookie:
def user2cookie(user, max_age):
    # build cookie string by: id-expires-sha1
    expires = str(int(time.time() + max_age))
    s = '%s-%s-%s-%s' % (user.id, user.passwd, expires, _COOKIE_KE
    L = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdiges
    return '-'.join(L)
```

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用middle在处理URL之前，把cookie解析出来，并将登录用户绑定到 `request` 对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

```
@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' % (request.method, request
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = yield from cookie2user(cookie_str)
            if user:
                logging.info('set current user: %s' % user.email)
                request.__user__ = user
        return (yield from handler(request))
    return auth

# 解密cookie:
@asyncio.coroutine
def cookie2user(cookie_str):
    '''
    Parse cookie and load user if cookie is valid.
    '''
    if not cookie_str:
        return None
    try:
        L = cookie_str.split('-')
        if len(L) != 3:
            return None
        uid, expires, sha1 = L
        if int(expires) < time.time():
            return None
        user = yield from User.find(uid)
        if user is None:
            return None
        s = '%s-%s-%s-%s' % (uid, user.passwd, expires, _COOKIE_KEY
        if sha1 != hashlib.sha1(s.encode('utf-8')).hexdigest():
            logging.info('invalid sha1')
            return None
        user.passwd = '******'
        return user
    except Exception as e:
```

```
        logging.exception(e)
        return None
```

这样，我们就完成了用户注册和登录的功能。

参考源码

day-10

Day 11 - 编写日志创建页

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog：

```
@post('/api/blogs')
def api_create_blog(request, *, name, summary, content):
    check_admin(request)
    if not name or not name.strip():
        raise APIValueError('name', 'name cannot be empty.')
    if not summary or not summary.strip():
        raise APIValueError('summary', 'summary cannot be empty.')
    if not content or not content.strip():
        raise APIValueError('content', 'content cannot be empty.')
    blog = Blog(user_id=request.__user__.id, user_name=request.__us
    yield from blog.save()
    return blog
```

编写后端Python代码不但很简单，而且非常容易测试，上面的API：`api_create_blog()` 本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

```
s = '<html><head><title>'
    + title
    + '</title></head><body>'
    + body
    + '</body></html>'
```

显然这种方式完全不具备可维护性。所以有第二种模板方式：

```
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    {{ body }}
</body>
</html>
```

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的MVVM：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

```
<script>
    var blog = {
        name: 'hello',
        summary: 'this is summary',
        content: 'this is content...'
    };
</script>
```

View是纯HTML：

```
<form action="/api/blogs" method="post">
    <input name="name">
    <input name="summary">
    <textarea name="content"></textarea>
    <button type="submit">OK</button>
</form>
```

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择Vue这个简单易用的MVVM框架来实现创建Blog的页面 `templates/manage_blog_edit.html` ：

```
{% extends '__base__.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}

<script>
var
    ID = '{{ id }}',
    action = '{{ action }}';
function initVM(blog) {
    var vm = new Vue({
        el: '#vm',
        data: blog,
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm').find('form');
                $form.postJSON(action, this.$data, function (err, r
                    if (err) {
                        $form.showFormError(err);
                    }
                    else {
                        return location.assign('/api/blogs/' + r.id
                    }
                });
            }
        }
```

```
    });
    $('#vm').show();
}
$(function () {
    if (ID) {
        getJSON('/api/blogs/' + ID, function (err, blog) {
            if (err) {
                return fatal(err);
            }
            $('#loading').hide();
            initVM(blog);
        });
    }
    else {
        $('#loading').hide();
        initVM({
            name: '',
            summary: '',
            content: ''
        });
    }
});
</script>

{% endblock %}

{% block content %}

    <div class="uk-width-1-1 uk-margin-bottom">
        <div class="uk-panel uk-panel-box">
            <ul class="uk-breadcrumb">
                <li><a href="/manage/comments">评论</a></li>
                <li><a href="/manage/blogs">日志</a></li>
                <li><a href="/manage/users">用户</a></li>
            </ul>
        </div>
    </div>

    <div id="error" class="uk-width-1-1">
    </div>
```

```
    <div id="loading" class="uk-width-1-1 uk-text-center">
        <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spi
    </div>

    <div id="vm" class="uk-width-2-3">
        <form v-on="submit: submit" class="uk-form uk-form-stacked'
            <div class="uk-alert uk-alert-danger uk-hidden"></div>
            <div class="uk-form-row">
                <label class="uk-form-label">标题:</label>
                <div class="uk-form-controls">
                    <input v-model="name" name="name" type="text" p
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">摘要:</label>
                <div class="uk-form-controls">
                    <textarea v-model="summary" rows="4" name="sumn
                </div>
            </div>
            <div class="uk-form-row">
                <label class="uk-form-label">内容:</label>
                <div class="uk-form-controls">
                    <textarea v-model="content" rows="16" name="cor
                </div>
            </div>
            <div class="uk-form-row">
                <button type="submit" class="uk-button uk-button-pr
                <a href="/manage/blogs" class="uk-button"><i class=
            </div>
        </form>
    </div>

{% endblock %}
```

初始化Vue时，我们指定3个参数：

el：根据选择器查找绑定的View，这里是 `#vm`，就是id为 `vm` 的DOM，对应的是一个 `<div>` 标签；

data：JavaScript对象表示的Model，我们初始化为 `{ name: '', summary: '', content: ''}` ；

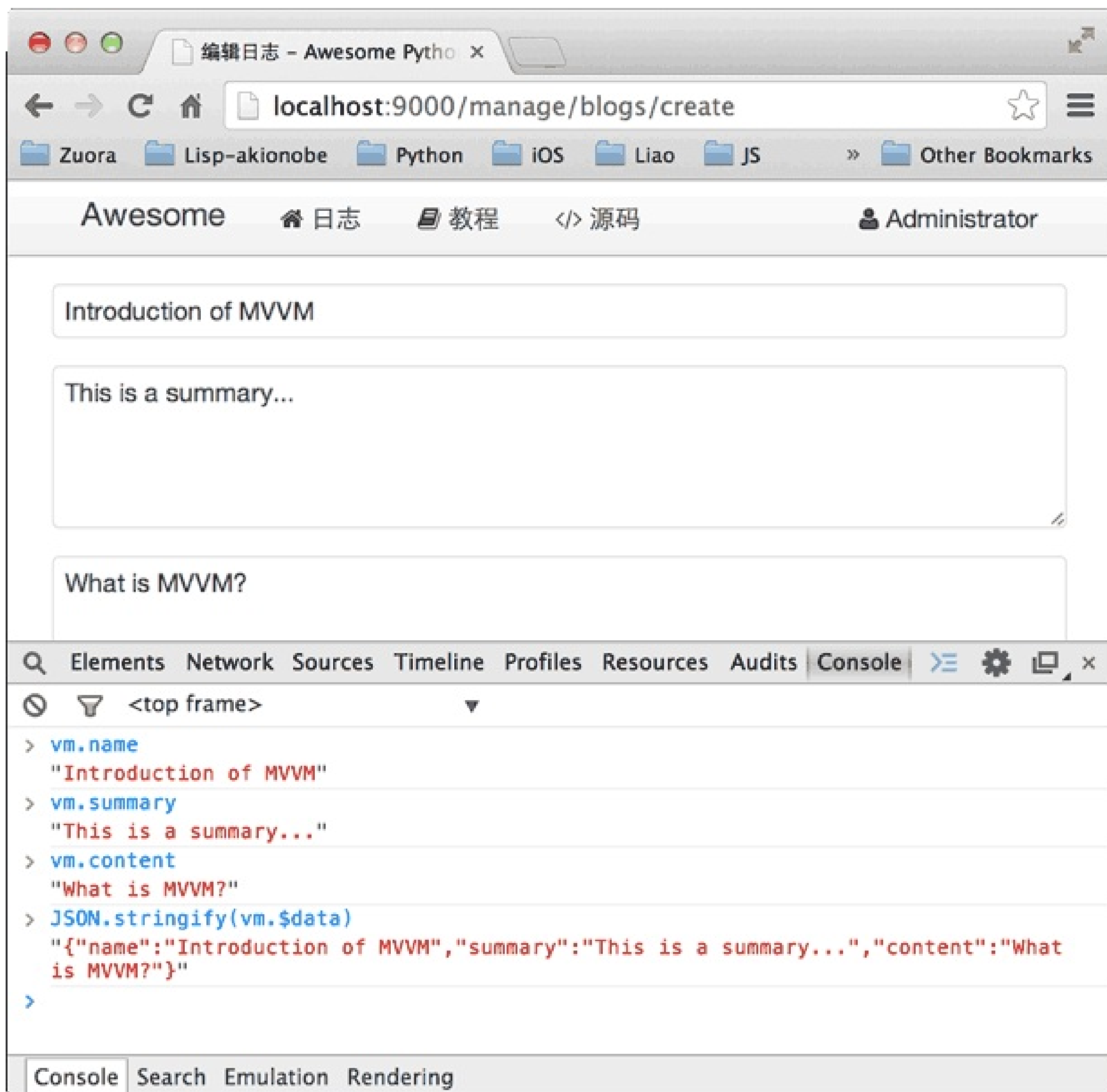
methods：View可以触发的JavaScript函数， `submit` 就是提交表单时触发的函数。

接下来，我们在 `<form>` 标签中，用几个简单的 `v-model` ，就可以让Vue把Model和View关联起来：

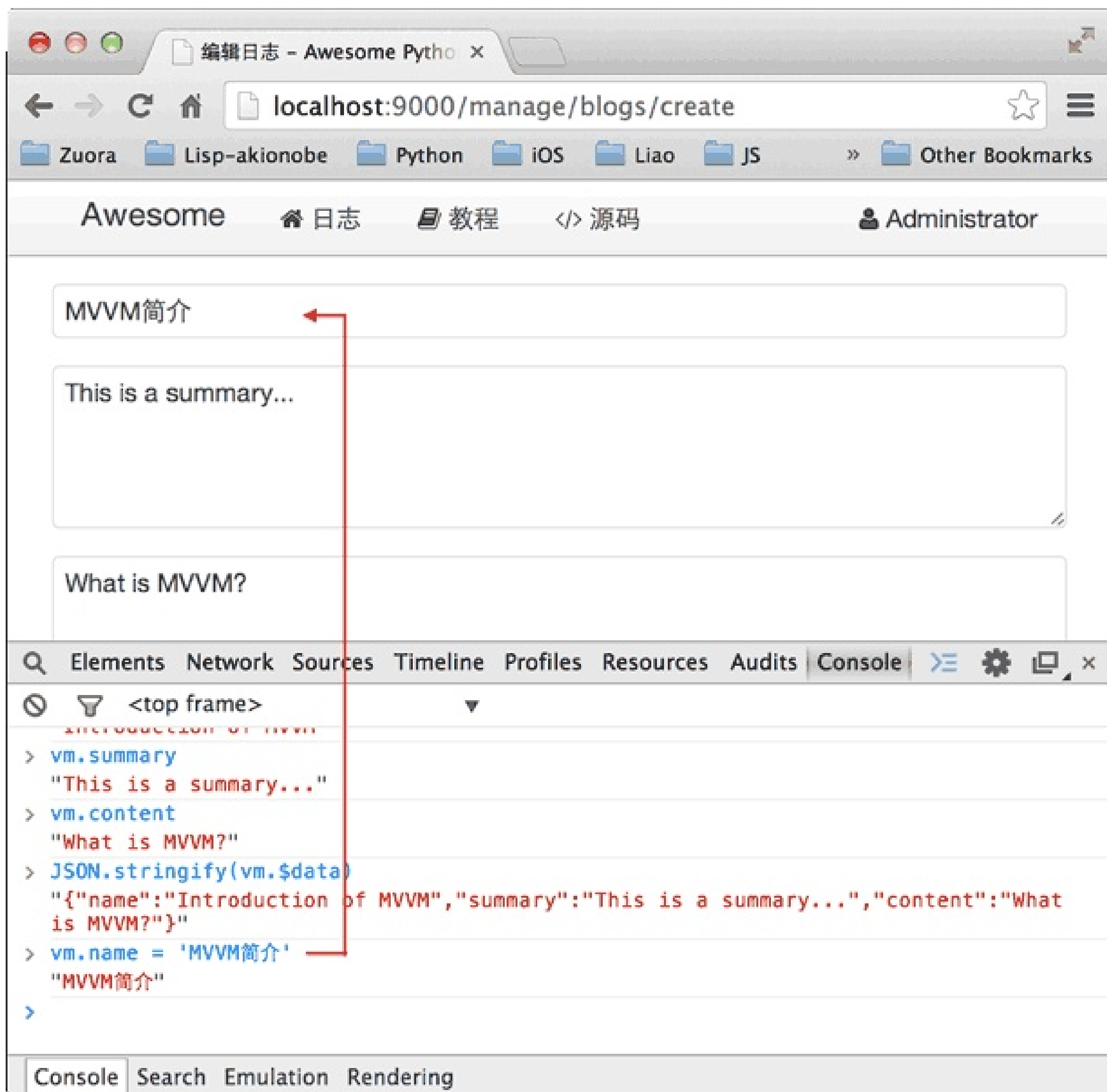
```
<!-- input的value和Model的name关联起来了 -->
<input v-model="name" class="uk-width-1-1">
```

Form表单通过 `<form v-on="submit: submit">` 把提交表单的事件关联到 `submit` 方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过 `vm.name` 访问单个属性，或者通过 `vm.$data` 访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入 `vm.name = 'MVVM简介'` ，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

参考源码

day-11

Day 12 - 编写日志列表页

MVVM模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来：

在 `apis.py` 中定义一个 `Page` 类用于存储分页信息：

```
class Page(object):

    def __init__(self, item_count, page_index=1, page_size=10):
        self.item_count = item_count
        self.page_size = page_size
        self.page_count = item_count // page_size + (1 if item_coun
        if (item_count == 0) or (page_index > self.page_count):
            self.offset = 0
            self.limit = 0
            self.page_index = 1
        else:
            self.page_index = page_index
            self.offset = self.page_size * (page_index - 1)
            self.limit = self.page_size
        self.has_next = self.page_index < self.page_count
        self.has_previous = self.page_index > 1

    def __str__(self):
        return 'item_count: %s, page_count: %s, page_index: %s, pag

    __repr__ = __str__
```

在 `handlers.py` 中实现API：

```
@get('/api/blogs')
def api_blogs(*, page='1'):
    page_index = get_page_index(page)
    num = yield from Blog.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, blogs=())
    blogs = yield from Blog.findAll(orderBy='created_at desc', lim:
    return dict(page=p, blogs=blogs)
```

管理页面：

```
@get('/manage/blogs')
def manage_blogs(*, page='1'):
    return {
        '__template__': 'manage_blogs.html',
        'page_index': get_page_index(page)
    }
```

模板页面首先通过API：`GET /api/blogs?page=?` 拿到Model：

```
{
    "page": {
        "has_next": true,
        "page_index": 1,
        "page_count": 2,
        "has_previous": false,
        "item_count": 12
    },
    "blogs": [...]
}
```

然后，通过Vue初始化MVVM：

```
<script>
function initVM(data) {
    var vm = new Vue({
        el: '#vm',
        data: {
            blogs: data.blogs,
            page: data.page
        },
        methods: {
            edit_blog: function (blog) {
                location.assign('/manage/blogs/edit?id=' + blog.id
            },
            delete_blog: function (blog) {
                if (confirm('确认要删除“' + blog.name + '”？删除后不可
                    postJSON('/api/blogs/' + blog.id + '/delete', f
                        if (err) {
                            return alert(err.message || err.error
                        }
                        refresh();
                    });
                }
            }
        }
    });
    $('#vm').show();
}
$(function() {
    getJSON('/api/blogs', {
        page: {{ page_index }}
    }, function (err, results) {
        if (err) {
            return fatal(err);
        }
        $('#loading').hide();
        initVM(results);
    });
});
</script>
```

View的容器是 `#vm` ，包含一个table，我们用 `v-repeat` 可以把Model的数组 `blogs` 直接变成多行的 `<tr>` ：

```
<div id="vm" class="uk-width-1-1">
    <a href="/manage/blogs/create" class="uk-button uk-button-prima

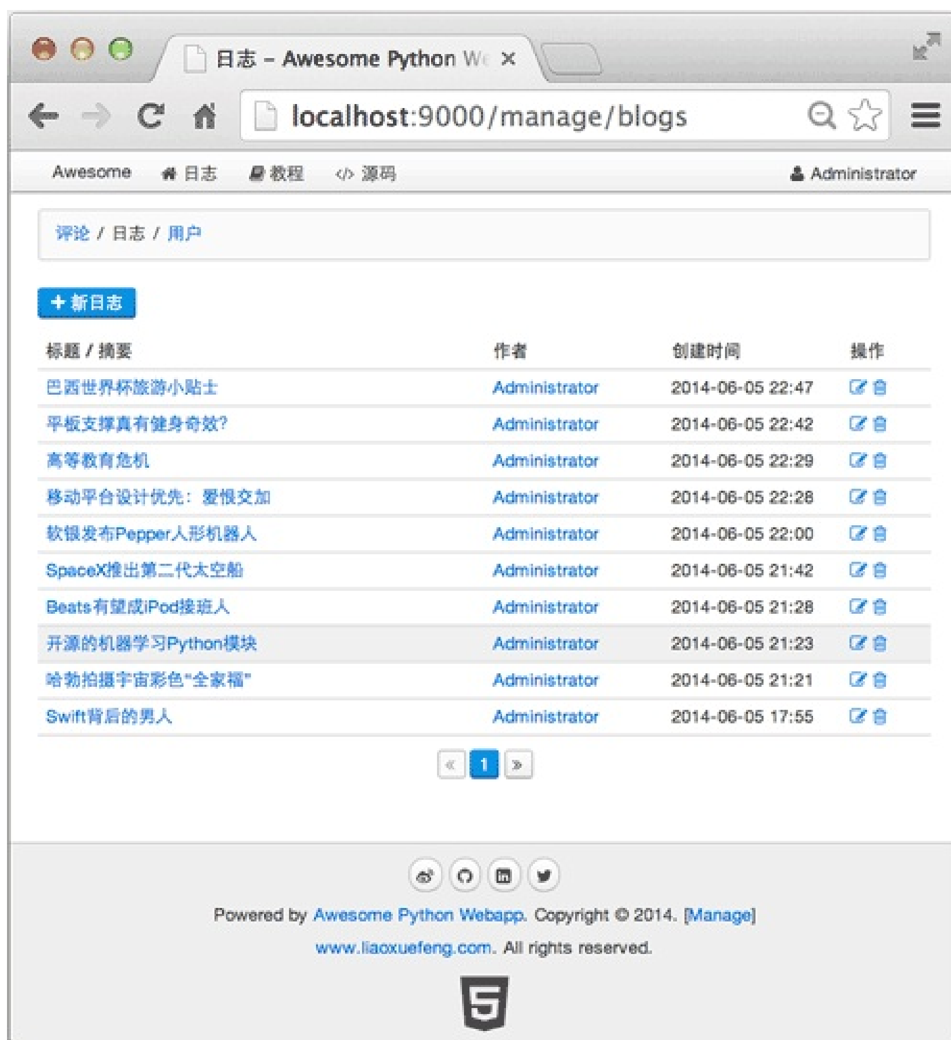
    <table class="uk-table uk-table-hover">
        <thead>
            <tr>
                <th class="uk-width-5-10">标题 / 摘要</th>
                <th class="uk-width-2-10">作者</th>
                <th class="uk-width-2-10">创建时间</th>
                <th class="uk-width-1-10">操作</th>
            </tr>
        </thead>
        <tbody>
            <tr v-repeat="blog: blogs" >
                <td>
                    <a target="_blank" v-attr="href: '/blog/'+blog.
                </td>
                <td>
                    <a target="_blank" v-attr="href: '/user/'+blog.
                </td>
                <td>
                    <span v-text="blog.created_at.toDateTime()"></s
                </td>
                <td>
                    <a href="#0" v-on="click: edit_blog(blog)"><i c
                    <a href="#0" v-on="click: delete_blog(blog)"><i
                </td>
            </tr>
        </tbody>
    </table>

    <div v-component="pagination" v-with="page"></div>
</div>
```

往Model的 `blogs` 数组中增加一个Blog元素，table就神奇地增加了一行；把 `blogs` 数组的某个元素删除，table就神奇地减少了一行。所有复杂的Model-View的映射逻辑全部由MVVM框架完成，我们只需要在HTML中写上 `v-repeat` 指令，就什么都不用管了。

可以把 `v-repeat="blog: blogs"` 看成循环代码，所以，可以在一个 `<tr>` 内部引用循环变量 `blog` 。 `v-text` 和 `v-attr` 指令分别用于生成文本和DOM节点属性。

完整的Blog列表页如下：



参考源码

day-12

Day 13 - 提升开发效率

现在，我们已经把一个Web App的框架完全搭建好了，从后端的API到前端的MVVM，流程已经跑通了。

在继续工作前，注意到每次修改Python代码，都必须在命令行先Ctrl-C停止服务器，再重启，改动才能生效。

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测 `www` 目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序 `pymonitor.py` ，让它启动 `wsgiapp.py` ，并时刻监控 `www` 目录下的代码改动，有改动时，先把当前 `wsgiapp.py` 进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库 `watchdog` 可以利用操作系统的API来监控目录文件的变化，并发送通知。我们先用 `pip` 安装：

```
$ pip3 install watchdog
```

利用 `watchdog` 接收文件变化的通知，如果是 `.py` 文件，就自动重启 `wsgiapp.py` 进程。

利用Python自带的 `subprocess` 实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
__author__ = 'Michael Liao'

import os, sys, time, subprocess

from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

def log(s):
    print('[Monitor] %s' % s)

class MyFileSystemEventHander(FileSystemEventHandler):

    def __init__(self, fn):
        super(MyFileSystemEventHander, self).__init__()
        self.restart = fn

    def on_any_event(self, event):
        if event.src_path.endswith('.py'):
            log('Python source file changed: %s' % event.src_path)
            self.restart()

command = ['echo', 'ok']
process = None

def kill_process():
    global process
    if process:
        log('Kill process [%s]...' % process.pid)
        process.kill()
        process.wait()
        log('Process ended with code %s.' % process.returncode)
        process = None

def start_process():
    global process, command
    log('Start process %s...' % ' '.join(command))
    process = subprocess.Popen(command, stdin=sys.stdin, stdout=sys

def restart_process():
```

```
    kill_process()
    start_process()

def start_watch(path, callback):
    observer = Observer()
    observer.schedule(MyFileSystemEventHander(restart_process), pat
    observer.start()
    log('Watching directory %s...' % path)
    start_process()
    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()

if __name__ == '__main__':
    argv = sys.argv[1:]
    if not argv:
        print('Usage: ./pymonitor your-script.py')
        exit(0)
    if argv[0] != 'python3':
        argv.insert(0, 'python3')
    command = argv
    path = os.path.abspath('.')
    start_watch(path, None)
```

一共70行左右的代码，就实现了Debug模式的自动重新加载。用下面的命令启动服务器：

```
$ python3 pymonitor.py wsgiapp.py
```

或者给 `pymonitor.py` 加上可执行权限，启动服务器：

```
$ ./pymonitor.py app.py
```

在编辑器中打开一个 `.py` 文件，修改后保存，看看命令行输出，是不是自动重启了服务器：

```
$ ./pymonitor.py app.py
[Monitor] Watching directory /Users/michael/Github/awesome-python3-
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-python3-webapp
[Monitor] Python source file changed: /Users/michael/Github/awesome
[Monitor] Kill process [2747]...
[Monitor] Process ended with code -9.
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-python3-webapp
```

现在，只要一保存代码，就可以刷新浏览器看到效果，大大提升了开发效率。

Day 14 - 完成Web App

在Web App框架和基本流程跑通后，剩下的工作全部是体力活了：在Debug开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：

把当前用户绑定到`request`上，并对URL`/manage/`进行拦截，检查当前用户是否是管理员身份：

```
@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' % (request.method, request
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = yield from cookie2user(cookie_str)
            if user:
                logging.info('set current user: %s' % user.email)
                request.__user__ = user
        if request.path.startswith('/manage/') and (request.__user_
            return web.HTTPFound('/signin')
        return (yield from handler(request))
    return auth
```

后端API包括：

- 获取日志：GET /api/blogs
- 创建日志：POST /api/blogs
- 修改日志：POST /api/blogs/:blog_id
- 删除日志：POST /api/blogs/:blog_id/delete
- 获取评论：GET /api/comments
- 创建评论：POST /api/blogs/:blog_id/comments

- 删除评论：POST /api/comments/:comment_id/delete
- 创建新用户：POST /api/users
- 获取用户：GET /api/users

管理页面包括：

- 评论列表页：GET /manage/comments
- 日志列表页：GET /manage/blogs
- 创建日志页：GET /manage/blogs/create
- 修改日志页：GET /manage/blogs/
- 用户列表页：GET /manage/users

用户浏览页面包括：

- 注册页：GET /register
- 登录页：GET /signin
- 注销页：GET /signout
- 首页：GET /
- 日志详情页：GET /blog/:blog_id

把所有的功能实现，我们第一个Web App就宣告完成！

参考源码

day-14

Day 15 - 部署Web App

作为一个合格的开发者，在本地环境下完成开发还远远不够，我们需要把Web App部署到远程服务器上，这样，广大用户才能访问到网站。

很多做开发的同学把部署这件事情看成是运维同学的工作，这种看法是完全错误的。首先，最近流行DevOps理念，就是说，开发和运维要变成一个整体。其次，运维的难度，其实跟开发质量有很大的关系。代码写得垃圾，运维再好也架不住天天挂掉。最后，DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务？那就得在开发时考虑到这一点。

下面，我们就来把awesome-python3-webapp部署到Linux服务器。

搭建Linux服务器

要部署到Linux，首先得有一台Linux服务器。要在公网上体验的同学，可以在Amazon的AWS申请一台EC2虚拟机（免费使用1年），或者使用国内的一些云服务器，一般都提供Ubuntu Server的镜像。想在本地部署的同学，请安装虚拟机，推荐使用VirtualBox。

我们选择的Linux服务器版本是Ubuntu Server 14.04 LTS，原因是apt太简单了。如果你准备使用其他Linux版本，也没有问题。

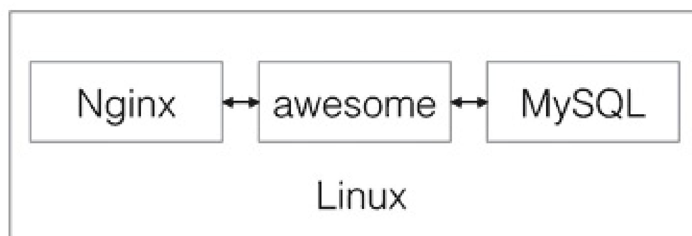
Linux安装完成后，请确保ssh服务正在运行，否则，需要通过apt安装：

```
$ sudo apt-get install openssh-server
```

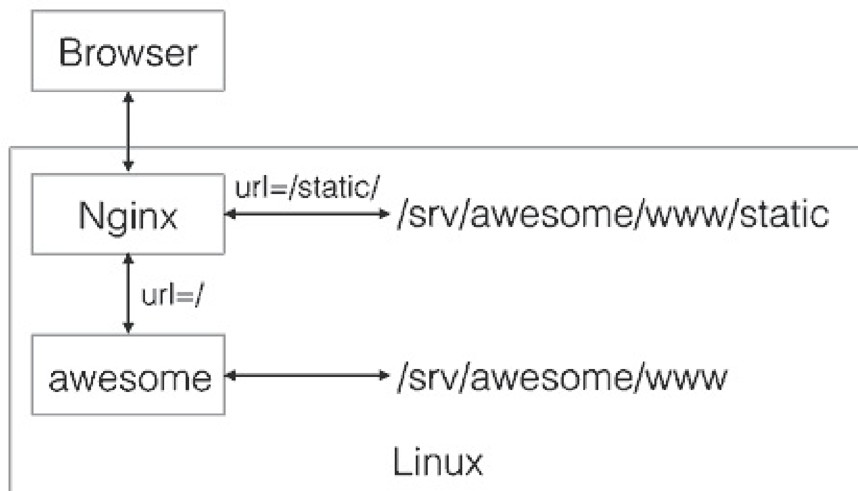
有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户的 `.ssh/authorized_keys` 中，这样，就可以通过证书实现无密码连接。

部署方式

利用Python自带的asyncio，我们已经编写了一个异步高性能服务器。但是，我们还需要一个高性能的Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。这个模型如下：



Nginx负责分发请求：



在服务器端，我们需要定义好部署的目录结构：

```
/
+- srv/
   +- awesome/       <-- Web App根目录
      +- www/        <-- 存放Python源码
      |  +- static/  <-- 存放静态资源文件
      +- log/        <-- 存放log
```

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把 `www` 作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本：

```
michael@ubuntu:/srv/awesome$ ls -l
total 32
drwxr-xr-x 2 www-data www-data 4096 Jun  5 17:38 log
lrwxrwxrwx 1 root     root       21 Jun  5 17:50 www -> www-14-06-05_17.56.16
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:26 www-14-06-05_15.26.45
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:31 www-14-06-05_15.31.03
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:32 www-14-06-05_15.32.39
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:34 www-14-06-05_17.39.28
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:35 www-14-06-05_17.41.22
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:39 www-14-06-05_17.45.57
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:50 www-14-06-05_17.56.16
michael@ubuntu:/srv/awesome$ _
```

而Nginx和gunicorn的配置文件只需要指向 `www` 目录即可。

Nginx可以作为服务进程直接启动，但gunicorn还不行，所以，Supervisor登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

- Nginx：高性能Web服务器+负责反向代理；
- Supervisor：监控服务进程的工具；
- MySQL：数据库服务。

在Linux服务器上用apt可以直接安装上述服务：

```
$ sudo apt-get install nginx supervisor python3 mysql-server
```

然后，再把我们自己的Web App用到的Python库安装了：

```
$ sudo pip3 install jinja2 aiomysql aiohttp
```

在服务器上创建目录 `/srv/awesome/` 以及相应的子目录。

在服务器上初始化MySQL数据库，把数据库初始化脚本 `schema.sql` 复制到服务器上执行：

```
$ mysql -u root -p < schema.sql
```

服务器端准备就绪。

部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。Fabric就是一个自动化部署工具。由于Fabric是用Python 2.x开发的，所以，部署脚本要用Python 2.7来编写，本机还必须安装Python 2.7版本。

要用Fabric部署，需要在本机（是开发机器，不是Linux服务器）安装Fabric：

```
$ easy_install fabric
```

Linux服务器上不需要安装Fabric，Fabric使用SSH直接登录服务器并执行部署命令。

下一步是编写部署脚本。Fabric的部署脚本叫 `fabfile.py` ，我们把它放到 `awesome-python-webapp` 的目录下，与 `www` 目录平级：

```
awesome-python-webapp/
+- fabfile.py
+- www/
+- ...
```

Fabric的脚本编写很简单，首先导入Fabric的API，设置部署时的变量：

```
# fabfile.py
import os, re
from datetime import datetime

# 导入Fabric API:
from fabric.api import *

# 服务器登录用户名:
env.user = 'michael'
# sudo用户为root:
env.sudo_user = 'root'
# 服务器地址，可以有多个，依次部署:
env.hosts = ['192.168.0.3']

# 服务器MySQL用户名和口令:
db_user = 'www-data'
db_password = 'www-data'
```

然后，每个Python函数都是一个任务。我们先编写一个打包的任务：

```
_TAR_FILE = 'dist-awesome.tar.gz'

def build():
    includes = ['static', 'templates', 'transwarp', 'favicon.ico',
    excludes = ['test', '.*', '*.pyc', '*.pyo']
    local('rm -f dist/%s' % _TAR_FILE)
    with lcd(os.path.join(os.path.abspath('.'), 'www')):
        cmd = ['tar', '--dereference', '-czvf', '../dist/%s' % _TAR
        cmd.extend(['--exclude=\'%s\'' % ex for ex in excludes])
        cmd.extend(includes)
        local(' '.join(cmd))
```

Fabric提供 `local('...')` 来运行本地命令， `with lcd(path)` 可以把当前命令的目录设定为 `lcd()` 指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要Cgywin环境。

在 `awesome-python-webapp` 目录下运行：

```
$ fab build
```

看看是否在 `dist` 目录下创建了 `dist-awesome.tar.gz` 的文件。

打包后，我们就可以继续编写 `deploy` 任务，把打包文件上传至服务器，解压，重置 `www` 软链接，重启相关服务：

```
_REMOTE_TMP_TAR = '/tmp/%s' % _TAR_FILE
_REMOTE_BASE_DIR = '/srv/awesome'

def deploy():
    newdir = 'www-%s' % datetime.now().strftime('%y-%m-%d_%H.%M.%S
    # 删除已有的tar文件:
    run('rm -f %s' % _REMOTE_TMP_TAR)
    # 上传新的tar文件:
    put('dist/%s' % _TAR_FILE, _REMOTE_TMP_TAR)
    # 创建新目录:
    with cd(_REMOTE_BASE_DIR):
        sudo('mkdir %s' % newdir)
    # 解压到新目录:
    with cd('%s/%s' % (_REMOTE_BASE_DIR, newdir)):
        sudo('tar -xzvf %s' % _REMOTE_TMP_TAR)
    # 重置软链接:
    with cd(_REMOTE_BASE_DIR):
        sudo('rm -f www')
        sudo('ln -s %s www' % newdir)
        sudo('chown www-data:www-data www')
        sudo('chown -R www-data:www-data %s' % newdir)
    # 重启Python服务和nginx服务器:
    with settings(warn_only=True):
        sudo('supervisorctl stop awesome')
        sudo('supervisorctl start awesome')
        sudo('/etc/init.d/nginx reload')
```

注意 `run()` 函数执行的命令是在服务器上运行， `with cd(path)` 和 `with lcd(path)` 类似，把当前目录在服务器端设置为 `cd()` 指定的目录。如果一个命令需要sudo权限，就不能用 `run()` ，而是用 `sudo()` 来执行。

配置Supervisor

上面让Supervisor重启awesome的命令会失败，因为我们还没有配置Supervisor呢。

编写一个Supervisor的配置文件 `awesome.conf` ，存放到 `/etc/supervisor/conf.d/` 目录下：

```
[program:awesome]

command     = /srv/awesome/www/app.py
directory   = /srv/awesome/www
user        = www-data
startsecs   = 3

redirect_stderr         = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups  = 10
stdout_logfile          = /srv/awesome/log/app.log
```

配置文件通过 `[program:awesome]` 指定服务名为 `awesome` ， `command` 指定启动 `app.py` 。

然后重启Supervisor后，就可以随时启动和停止Supervisor管理的服务了：

```
$ sudo supervisorctl reload
$ sudo supervisorctl start awesome
$ sudo supervisorctl status
awesome                RUNNING    pid 1401, uptime 5:01:34
```

配置Nginx

Supervisor只负责运行gunicorn，我们还需要配置Nginx。把配置文件 `awesome` 放到 `/etc/nginx/sites-available/` 目录下：

```
server {
    listen      80; # 监听80端口

    root       /srv/awesome/www;
    access_log /srv/awesome/log/access_log;
    error_log  /srv/awesome/log/error_log;

    # server_name awesome.liaoxuefeng.com; # 配置域名

    # 处理静态文件/favicon.ico:
    location /favicon.ico {
        root /srv/awesome/www;
    }

    # 处理静态资源:
    location ~ ^\/static\/.*$ {
        root /srv/awesome/www;
    }

    # 动态请求转发到9000端口:
    location / {
        proxy_pass       http://127.0.0.1:9000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for
    }
}
```

然后在 `/etc/nginx/sites-enabled/` 目录下创建软链接：

```
$ pwd
/etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/awesome .
```

让Nginx重新加载配置文件，不出意外，我们的 `awesome-python3-webapp` 应该正常运行：

```
$ sudo /etc/init.d/nginx reload
```

如果有任何错误，都可以在 `/srv/awesome/log` 下查找Nginx和App本身的log。如果Supervisor启动时报错，可以在 `/var/log/supervisor` 下查看Supervisor的log。

如果一切顺利，你可以在浏览器中访问Linux服务器上的 `awesome-python3-webapp` 了：



如果在开发环境更新了代码，只需要在命令行执行：

```
$ fab build
$ fab deploy
```

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

友情链接

嫌国外网速慢的童鞋请移步网易和搜狐的镜像站点：

http://mirrors.163.com/

http://mirrors.sohu.com/

参考源码

day-15

Day 16 - 编写移动App

网站部署上线后，还缺点啥呢？

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动App，出门根本不好意思跟人打招呼。

所以， `awesome-python3-webapp` 必须得有一个移动App版本！

开发iPhone版本

我们首先来看看如何开发iPhone App。前置条件：一台Mac电脑，安装XCode和最新的iOS SDK。

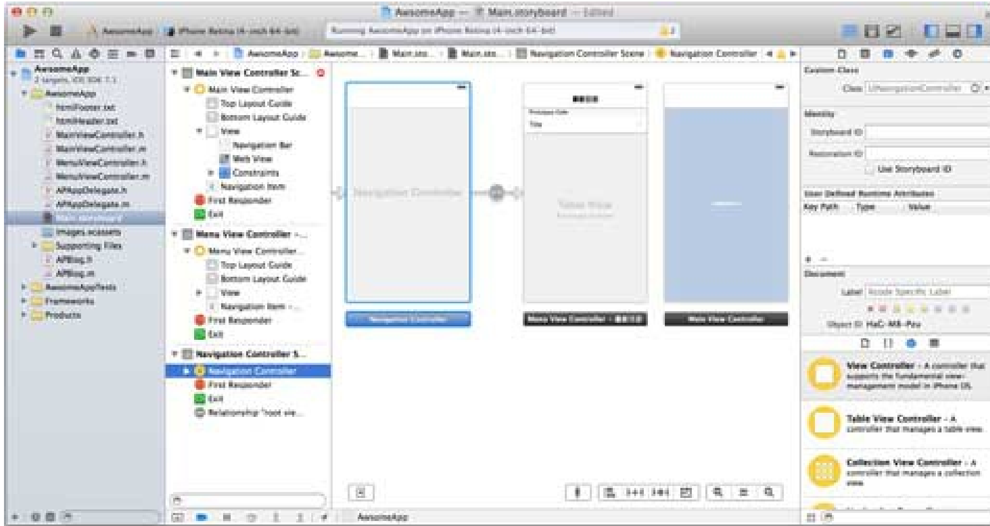
在使用MVVM编写前端页面时，我们就能感受到，用REST API封装网站后台的功能，不但能清晰地分离前端页面和后台逻辑，现在这个好处更加明显，移动App也可以通过REST API从后端拿到数据。

我们来设计一个简化版的iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容：



只需要调用API： `/api/blogs` 。

在XCode中完成App编写：



由于我们的教程是Python，关于如何开发iOS，请移步Develop Apps for iOS。

点击下载iOS App源码。

如何编写Android App？这个当成作业了。

参考源码

day-16

FAQ

常见问题

本节列出常见的一些问题。

如何获取当前路径

当前路径可以用 `'.'` 表示，再用 `os.path.abspath()` 将其转换为绝对路径：

```
# -*- coding:utf-8 -*-
# test.py

import os

print(os.path.abspath('.'))
```

运行结果：

```
$ python3 test.py
/Users/michael/workspace/testing
```

如何获取当前模块的文件名

可以通过特殊变量 `__file__` 获取：

```
# -*- coding:utf-8 -*-
# test.py

print(__file__)
```

输出：

```
$ python3 test.py
test.py
```

如何获取命令行参数

可以通过 `sys` 模块的 `argv` 获取：

```
# -*- coding:utf-8 -*-
# test.py

import sys

print(sys.argv)
```

输出：

```
$ python3 test.py -a -s "Hello world"
['test.py', '-a', '-s', 'Hello world']
```

`argv` 的第一个元素永远是命令行执行的 `.py` 文件名。

如何获取当前Python命令的可执行文件路径

`sys` 模块的 `executable` 变量就是Python命令可执行文件的路径：

```
# -*- coding:utf-8 -*-
# test.py

import sys

print(sys.executable)
```

在Mac下的结果：

```
$ python3 test.py
/usr/local/opt/python3/bin/python3.4
```

期末总结

终于到了期末总结的时刻了！

经过一段时间的学习，相信你对Python已经初步掌握。一开始，可能觉得Python上手很容易，可是越往后学，会越困难，有的时候，发现理解不了代码，这时，不妨停下来思考一下，先把概念搞清楚，代码自然就明白了。

Python非常适合初学者用来进入计算机编程领域。Python属于非常高级的语言，掌握了这门高级语言，就对计算机编程的核心思想——抽象有了初步理解。如果希望继续深入学习计算机编程，可以学习C、JavaScript、Lisp等不同类型的语言，只有多掌握不同领域的语言，有比较才更有收获。



Git教程



史上最浅显易懂的Git教程！

为什么要编写这个教程？因为我在学习Git的过程中，买过书，也在网上Google了一堆Git相关的文章和教程，但令人失望的是，这些教程不是难得令人发指，就是简单得一笔带过，或者，只支离破碎地介绍Git的某几个命令，还有直接从Git手册粘贴帮助文档的，总之，初学者很难找到一个由浅入深，学完后能立刻上手的Git教程。

既然号称史上最浅显易懂的Git教程，那这个教程有什么让你怦然心动的特点呢？

首先，本教程绝对面向初学者，没有接触过版本控制概念的读者也可以轻松入门，不必担心起步难度；

其次，本教程实用性超强，边学边练，一点也不觉得枯燥。而且，你所学的Git命令是“充分且必要”的，掌握了这些东西，你就可以通过Git轻松地完成你的工作。

文字+图片还看不明白？有视频！！！

本教程只会让你成为Git用户，不会让你成为Git专家。很多Git命令只有那些专家才明白（事实上我也不明白，因为我不是Git专家），但我保证这些命令可能你一辈子都不会用到。既然Git是一个工具，就没必要把时间浪费在那些“高级”但几乎永远不会用到的命令上。一旦你真的非用不可了，到时候再自行Google或者请教专家也未迟。

如果你是一个开发人员，想用上这个世界上目前最先进的分布式版本控制系统，那么，赶快开始学习吧！

关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Visual Basic/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub](https://github.com/michaelliao)，欢迎微博交流：@廖雪峰

Git简介

Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

Git有什么特点？简单来说就是：高端大气上档次！

那什么是版本控制系统？

如果你用Microsoft Word写过长篇大论，那你一定有这样的经历：

想删除一个段落，又怕将来想恢复找不回来怎么办？有办法，先把当前文件“另存为……”一个新的Word文件，再接着改，改到一定程度，再“另存为……”一个新文件，这样一直改下去，最后你的Word文档变成了这样：



过了一周，你想找回被删除的文字，但是已经记不清删除前保存在哪个文件里了，只好一个一个文件去找，真麻烦。

看着一堆乱七八糟的文件，想保留最新的一个，然后把其他的删掉，又怕哪天会用上，还不敢删，真郁闷。

更要命的是，有些部分需要你的财务同事帮助填写，于是你把文件Copy到U盘里给她（也可能通过Email发送一份给她），然后，你继续修改Word文件。一天后，同事再把Word文件传给你，此时，你必须想想，发给她之后到你收到她的文件期间，你作了哪些改动，得把你的改动和她的部分合并，真困难。

于是你想，如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以，岂不是很方便？

这个软件用起来就应该像这个样子，能记录每次文件的改动：

版本	用户	说明	日期
1	张三	删除了软件服务条款5	7/12 10:38
2	张三	增加了License人数限制	7/12 18:09
3	李四	财务部门调整了合同金额	7/13 9:51
4	张三	延长了免费升级周期	7/14 15:17

这样，你就结束了手动管理多个“版本”的史前时代，进入到版本控制的20世纪。

Git的诞生

很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

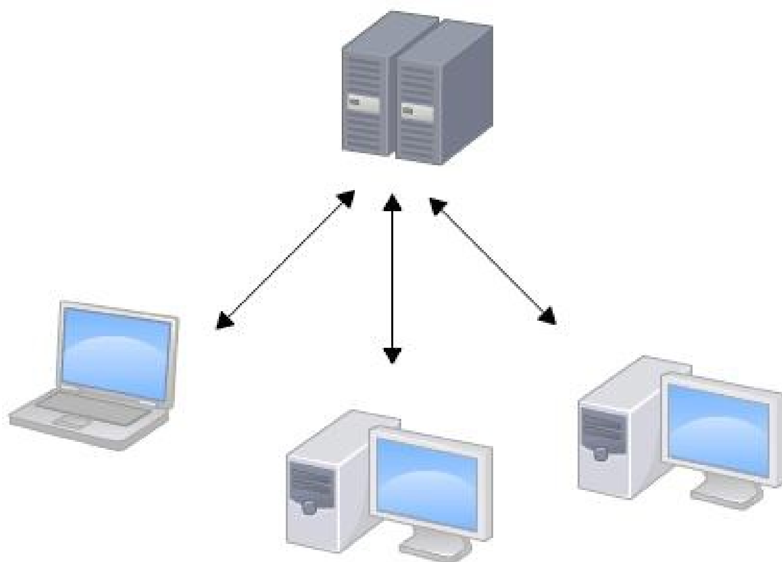
Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

集中式vs分布式

Linus一直痛恨的CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。

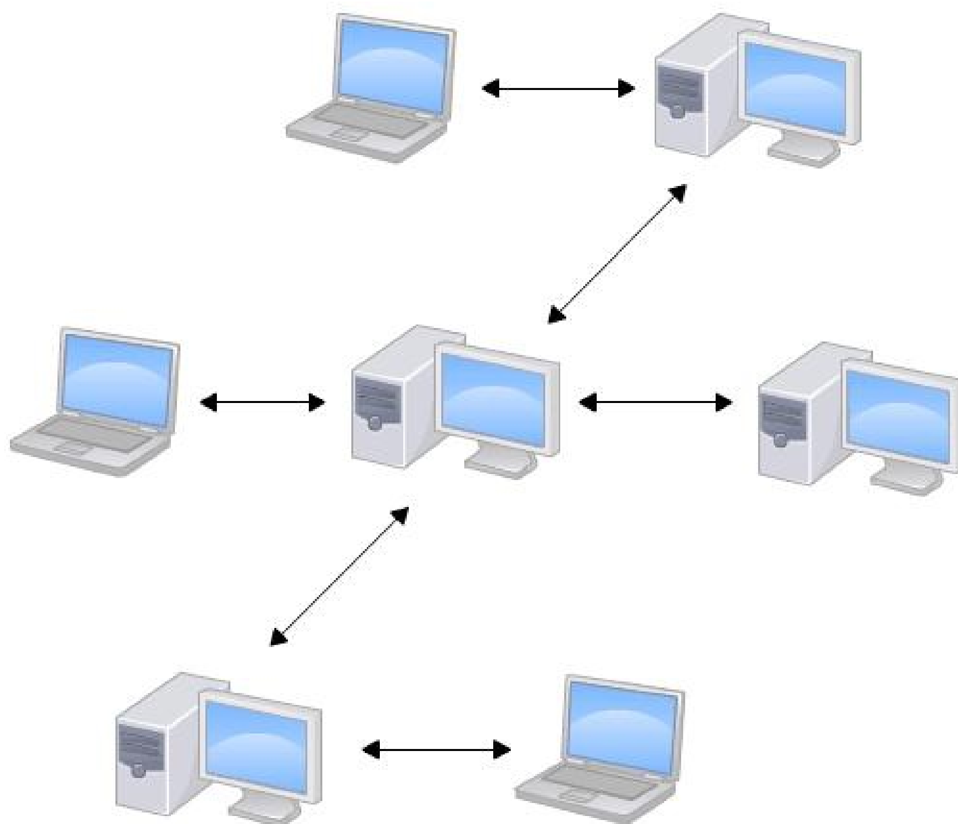


集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。



当然，Git的优势不单是不必联网这么简单，后面我们还会看到Git极其强大的分支管理，把SVN等远远抛在了后面。

CVS作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人在用。由于CVS自身设计的问题，会造成提交文件不完整，版本库莫名其妙损坏的情况。同样是开源而且免费的SVN修正了CVS的一些稳定性问题，是目前用得最多的集中式版本库控制系统。

除了免费的外，还有收费的集中式版本控制系统，比如IBM的ClearCase（以前是Rational公司的，被IBM收购了），特点是安装比Windows还大，运行比蜗牛还慢，能用ClearCase的一般是世界500强，他们有个共同的特点是财大气粗，或者人傻钱多。

微软自己也有一个集中式版本控制系统叫VSS，集成在Visual Studio中。由于其反人类的设计，连微软自己都不好意思用了。

分布式版本控制系统除了Git以及促使Git诞生的BitKeeper外，还有类似Git的Mercurial和Bazaar等。这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是Git！

安装Git

最早Git是在Linux上开发的，很长一段时间内，Git也只能在Linux和Unix系统上跑。不过，慢慢地有人把它移植到了Windows上。现在，Git可以在Linux、Unix、Mac和Windows这几大平台上正常运行了。

要使用Git，第一步当然是安装Git了。根据你当前使用的平台来阅读下面的文字：

在Linux上安装Git

首先，你可以试着输入 `git` ，看看系统有没有安装Git：

```
$ git
The program 'git' is currently not installed. You can install it by
sudo apt-get install git
```

像上面的命令，有很多Linux会友好地告诉你Git没有安装，还会告诉你如何安装Git。

如果你碰巧用Debian或Ubuntu Linux，通过一条 `sudo apt-get install git` 就可以直接完成Git的安装，非常简单。

http://michaelliao.gitcafe.io/video/git-apt-install.mp4

老一点的Debian或Ubuntu Linux，要把命令改为 `sudo apt-get install git-core` ，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫 `git-core` 了。由于Git名气实在太大，后来就把GNU Interactive Tools改成 `gnuit` ， `git-core` 正式改为 `git` 。

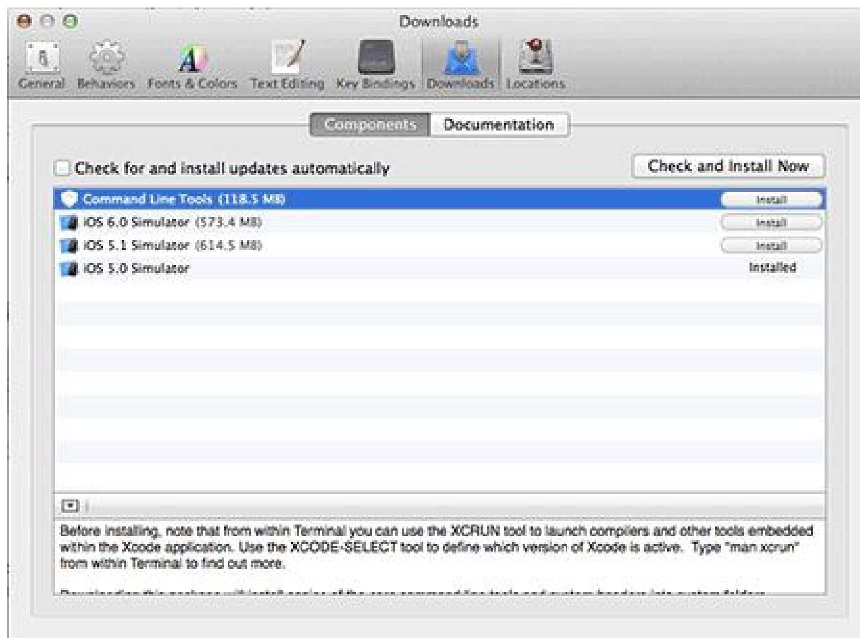
如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入： `./config` ， `make` ， `sudo make install` 这几个命令安装就好了。

在Mac OS X上安装Git

如果你正在使用Mac做开发，有两种安装Git的方法。

一是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：http://brew.sh/。

第二种方法更简单，也是推荐的方法，就是直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。



Xcode是Apple官方IDE，功能非常强大，是开发Mac和iOS App的必选装备，而且是免费的！

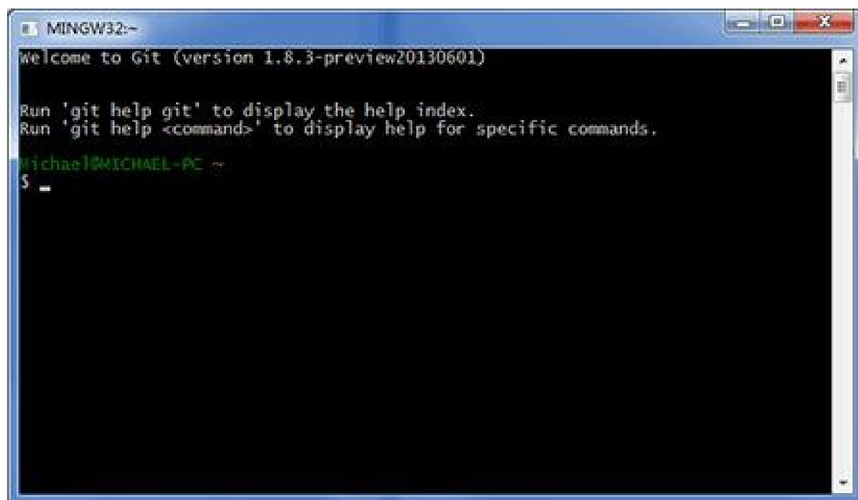
在Windows上安装Git

实话实说，Windows是最烂的开发平台，如果不是开发Windows游戏或者在IE里调试页面，一般不推荐用Windows。不过，既然已经上了微软的贼船，也是有办法安装Git的。

Windows下要使用很多Linux/Unix的工具时，需要Cygwin这样的模拟环境，Git也一样。Cygwin的安装和配置都比较复杂，就不建议你折腾了。不过，有高人已经把模拟环境和Git都打包好了，名叫msysgit，只需要下载一个单独的exe安装程序，其他什么也不用装，绝对好用。

msysgit是Windows版的Git，从http://msysgit.github.io/下载，然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！



安装完成后，还需要最后一步设置，在命令行输入：

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意 `git config` 命令的 `--global` 参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

创建版本库

什么是版本库呢？版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

```
$ mkdir learngit
$ cd learngit
$ pwd
/Users/michael/learngit
```

`pwd` 命令用于显示当前目录。在我的Mac上，这个仓库位于 `/Users/michael/learngit` 。

如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

第二步，通过 `git init` 命令把这个目录变成Git可以管理的仓库：

```
$ git init
Initialized empty Git repository in /Users/michael/learngit/.git/
```

瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个 `.git` 的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

如果你没有看到 `.git` 目录，那是因为这个目录默认是隐藏的，用 `ls -ah` 命令就可以看见。

http://michaelliao.gitcafe.io/video/git-init.mp4

也不一定必须在空目录下创建Git仓库，选择一个已经有东西的目录也是可以的。不过，不建议你使用自己正在开发的公司项目来学习Git，否则造成的一切后果概不负责。

把文件添加到版本库

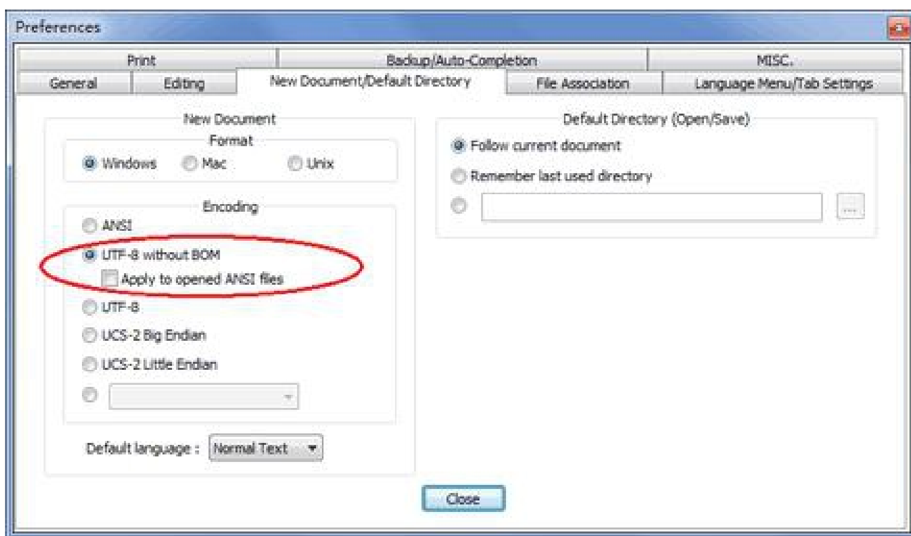
首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

使用Windows的童鞋要特别注意：

千万不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：



言归正传，现在我们编写一个 `readme.txt` 文件，内容如下：

```
Git is a version control system.
Git is free software.
```

一定要放到 `learngit` 目录下（子目录也行），因为这是一个Git仓库，放到其他地方Git再厉害也找不到这个文件。

和把大象放到冰箱需要3步相比，把一个文件放到Git仓库只需要两步。

第一步，用命令 `git add` 告诉Git，把文件添加到仓库：

```
$ git add readme.txt
```

执行上面的命令，没有任何显示，这就对了，Unix的哲学是“没有消息就是好消息”，说明添加成功。

第二步，用命令 `git commit` 告诉Git，把文件提交到仓库：

```
$ git commit -m "wrote a readme file"
[master (root-commit) cb926e7] wrote a readme file
 1 file changed, 2 insertions(+)
 create mode 100644 readme.txt
```

http://michaelliao.gitcafe.io/video/add-and-commit.mp4

简单解释一下`git commit`命令，`-m`后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

嫌麻烦不想输入`-m "xxx"`行不行？确实有办法可以这么干，但是强烈不建议你这么干，因为输入说明对自己对别人阅读都很重要。实在不想输入说明的童鞋请自行Google，我不告诉你这个参数。

`git commit`命令执行成功后会告诉你，1个文件被改动（我们新添加的readme.txt文件），插入了两行内容（readme.txt有两行内容）。

为什么Git添加文件需要`add`，`commit`一共两步呢？因为`commit`可以一次提交很多文件，所以你可以多次`add`不同的文件，比如：

```
$ git add file1.txt
$ git add file2.txt file3.txt
$ git commit -m "add 3 files."
```

小结

现在总结一下今天学的两点内容：

初始化一个Git仓库，使用`git init`命令。

添加文件到Git仓库，分两步：

- 第一步，使用命令`git add <file>`，注意，可反复多次使用，添加多个文件；
- 第二步，使用命令`git commit`，完成。

时光机穿梭

我们已经成功地添加并提交了一个readme.txt文件，现在，是时候继续工作了，于是，我们继续修改readme.txt文件，改成如下内容：

```
Git is a distributed version control system.
Git is free software.
```

现在，运行`git status`命令看看结果：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#    modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

`git status`命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

虽然Git告诉我们readme.txt被修改了，但如果能看看具体修改了什么内容，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的readme.txt，所以，需要用`git diff`这个命令看看：

```
$ git diff readme.txt
diff --git a/readme.txt b/readme.txt
index 46d49bf..9247db6 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
```

`git diff` 顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第一行添加了一个“distributed”单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是 `git add` ：

```
$ git add readme.txt
```

同样没有任何输出。在执行第二步 `git commit` 之前，我们再运行 `git status` 看看当前仓库的状态：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

`git status` 告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心地提交了：

```
$ git commit -m "add distributed"
[master ea34578] add distributed
 1 file changed, 1 insertion(+), 1 deletion(-)
```

提交后，我们再用`git status`命令看看仓库的当前状态：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working directory clean）的。

http://michaelliao.gitcafe.io/video/git-diff-status.mp4

小结

- 要随时掌握工作区的状态，使用`git status`命令。
- 如果`git status`告诉你有文件被修改过，用`git diff`可以查看修改内容。

版本回退

现在，你已经学会了修改文件，然后把修改提交到Git版本库，现在，再练习一次，修改readme.txt文件如下：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
```

然后尝试提交：

```
$ git add readme.txt
$ git commit -m "append GPL"
[master 3628164] append GPL
 1 file changed, 1 insertion(+), 1 deletion(-)
```

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为 `commit` 。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个 `commit` 恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在，我们回顾一下readme.txt文件一共有几个版本被提交到Git仓库里了：

版本1：wrote a readme file

```
Git is a version control system.
Git is free software.
```

版本2：add distributed

```
Git is a distributed version control system.
Git is free software.
```

版本3：append GPL

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
```

当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用 `git log` 命令查看：

```
$ git log
commit 3628164fb26d48395383f8f31179f24e0882e1e0
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 15:11:49 2013 +0800

    append GPL

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 14:53:12 2013 +0800

    add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 19 17:51:55 2013 +0800

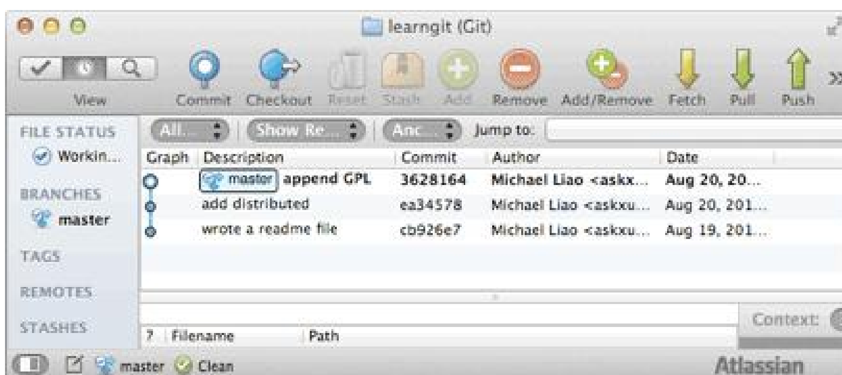
    wrote a readme file
```

`git log` 命令显示从最近到最远的提交日志，我们可以看到3次提交，最近的一次是 `append GPL` ，上一次是 `add distributed` ，最早的一次是 `wrote a readme file` 。 如果嫌输出信息太多，看得眼花缭乱的，可以试试加上 `--pretty=oneline` 参数：

```
$ git log --pretty=oneline
3628164fb26d48395383f8f31179f24e0882e1e0 append GPL
ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed
cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file
```

需要友情提示的是，你看到的一大串类似 `3628164...882e1e0` 的是 `commit id` （版本号），和SVN不一样，Git的 `commit id` 不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的 `commit id` 和我的肯定不一样，以你自己的为准。为什么 `commit id` 需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线：



好了，现在我们启动时光穿梭机，准备把readme.txt回退到上一个版本，也就是“add distributed”的那个版本，怎么做呢？

首先，Git必须知道当前版本是哪个版本，在Git中，用 `HEAD` 表示当前版本，也就是最新的提交 `3628164...882e1e0` （注意我的提交ID和你的肯定不一样），上一个版本就是 `HEAD^` ，上上一个版本就是 `HEAD^^` ，当然往上100个版本写100个 `^` 比较容易数不过来，所以写成 `HEAD~100` 。

现在，我们要把当前版本“append GPL”回退到上一个版本“add distributed”，就可以使用 `git reset` 命令：

```
$ git reset --hard HEAD^
HEAD is now at ea34578 add distributed
```

`--hard` 参数有啥意义？这个后面再讲，现在你先放心使用。

看看readme.txt的内容是不是版本 `add distributed` ：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software.
```

果然。

还可以继续回退到上一个版本 `wrote a readme file` ，不过且慢，然我们用 `git log` 再看看现在版本库的状态：

```
$ git log
commit ea34578d5496d7dd233c827ed32a8cd576c5ee85
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 14:53:12 2013 +0800

    add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 19 17:51:55 2013 +0800

    wrote a readme file
```

最新的那个版本 `append GPL` 已经看不到了！好比你从21世纪坐时光穿梭机来到了19世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个 `append GPL` 的 `commit id` 是 `3628164...` ，于是就可以指定回到未来的某个版本：

```
$ git reset --hard 3628164
HEAD is now at 3628164 append GPL
```

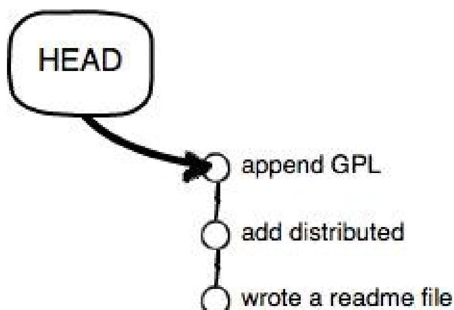
版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看readme.txt的内容：

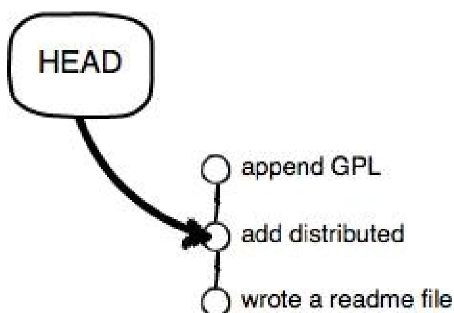
```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
```

果然，我胡汉三又回来了。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的 `HEAD` 指针，当你回退版本的时候，Git仅仅是把HEAD从指向 `append GPL` ：



改为指向 `add distributed` ：



然后顺便把工作区的文件更新了。所以你让 `HEAD` 指向哪个版本号，你就把当前版本定位在哪。

http://michaelliao.gitcafe.io/video/git-reset.mp4

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的 `commit id` 怎么办？

在Git中，总是有后悔药可以吃的。当你用 `$ git reset --hard HEAD^` 回退到 `add distributed` 版本时，再想恢复到 `append GPL` ，就必须找到 `append GPL` 的commit id。Git提供了一个命令 `git reflog` 用来记录你的每一次命令：

```
$ git reflog
ea34578 HEAD@{0}: reset: moving to HEAD^
3628164 HEAD@{1}: commit: append GPL
ea34578 HEAD@{2}: commit: add distributed
cb926e7 HEAD@{3}: commit (initial): wrote a readme file
```

终于舒了口气，第二行显示 `append GPL` 的commit id是 `3628164` ，现在，你又可以乘坐时光机回到未来了。

http://michaelliao.gitcafe.io/video/git-reflog-reset.mp4

小结

现在总结一下：

- `HEAD` 指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令 `git reset --hard commit_id` 。
- 穿梭前，用 `git log` 可以查看提交历史，以便确定要回退到哪个版本。
- 要重返未来，用 `git reflog` 查看命令历史，以便确定要回到未来的哪个版本。

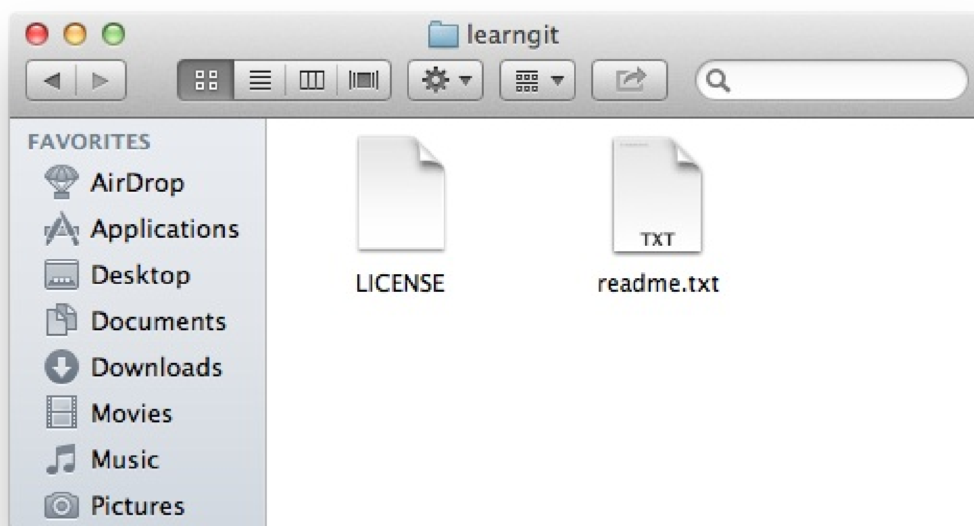
工作区和暂存区

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

先来看名词解释。

工作区（Working Directory）

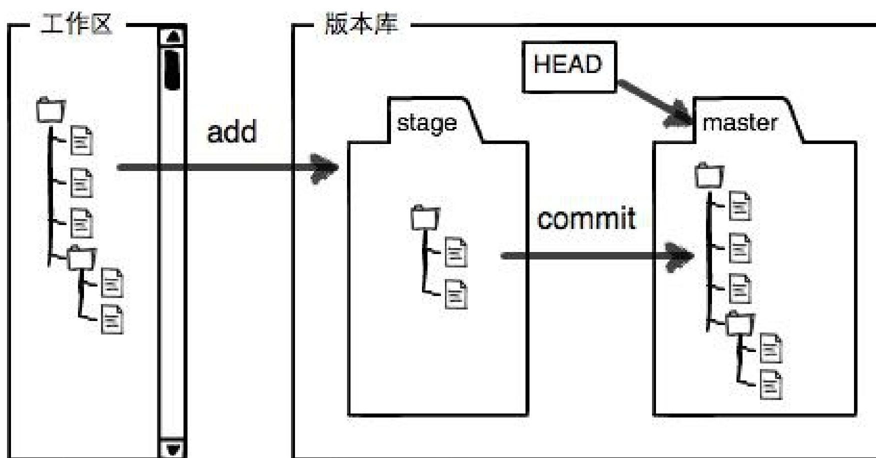
就是你在电脑里能看到的目录，比如我的 `learngit` 文件夹就是一个工作区：



版本库（Repository）

工作区有一个隐藏目录 `.git` ，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支 `master` ，以及指向 `master` 的一个指针叫 `HEAD` 。



分支和 `HEAD` 的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用 `git add` 把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用 `git commit` 提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个 `master` 分支，所以，现在， `git commit` 就是往 `master` 分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

俗话说，实践出真知。现在，我们再练习一遍，先对 `readme.txt` 做个修改，比如加上一行内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
```

然后，在工作区新增一个 `LICENSE` 文本文件（内容随便写）。

先用 `git status` 查看一下状态：

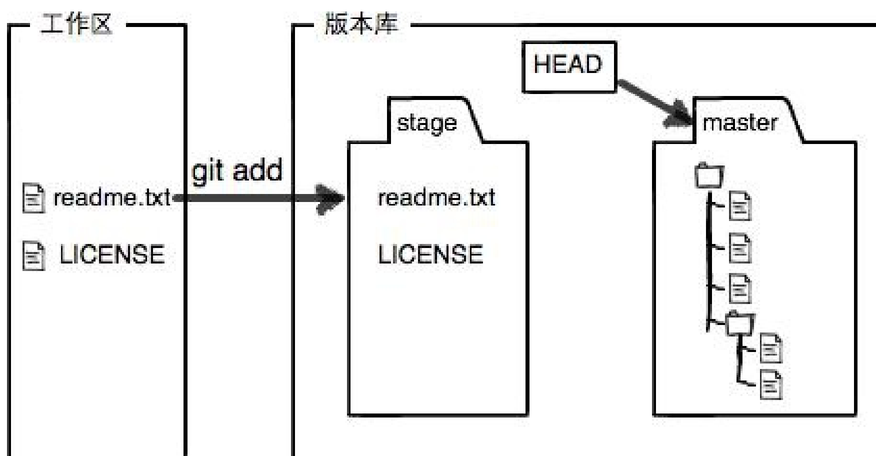
```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       LICENSE
no changes added to commit (use "git add" and/or "git commit -a")
```

Git非常清楚地告诉我们， `readme.txt` 被修改了，而 `LICENSE` 还从来没有被添加过，所以它的状态是 `Untracked` 。

现在，使用两次命令 `git add` ，把 `readme.txt` 和 `LICENSE` 都添加后，用 `git status` 再查看一下：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   LICENSE
#       modified:   readme.txt
#
```

现在，暂存区的状态就变成这样了：



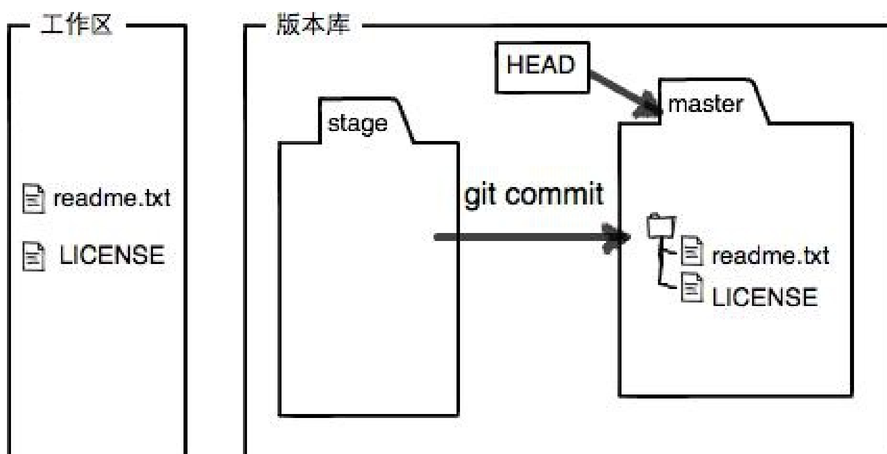
所以，`git add`命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行`git commit`就可以一次性把暂存区的所有修改提交到分支。

```
$ git commit -m "understand how stage works"
[master 27c9860] understand how stage works
 2 files changed, 675 insertions(+)
 create mode 100644 LICENSE
```

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

现在版本库变成了这样，暂存区就没有任何内容了：



小结

暂存区是Git非常重要的概念，弄明白了暂存区，就弄明白了Git的很多操作到底干了什么。

没弄明白暂存区是怎么回事的童鞋，请向上滚动页面，再看一次。

管理修改

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

为什么说Git管理的是修改，而不是文件呢？我们还是做实验。第一步，对readme.txt做一个修改，比如加一行内容：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes.
```

然后，添加：

```
$ git add readme.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

然后，再修改readme.txt：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
```

提交：

```
$ git commit -m "git tracks changes"
[master d4f25b6] git tracks changes
 1 file changed, 1 insertion(+)
```

提交后，再看看状态：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

第一次修改 -> `git add` -> 第二次修改 -> `git commit`

你看，我们前面讲了，Git管理的是修改，当你用 `git add` 命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以， `git commit` 只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用 `git diff HEAD -- readme.txt` 命令可以查看工作区和版本库里面最新版本的区别：

```
$ git diff HEAD -- readme.txt
diff --git a/readme.txt b/readme.txt
index 76d770f..a9c5755 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,4 +1,4 @@
 Git is a distributed version control system.
 Git is free software distributed under the GPL.
 Git has a mutable index called stage.
-Git tracks changes.
+Git tracks changes of files.
```

可见，第二次修改确实没有被提交。

http://michaelliao.gitcafe.io/video/git-add-changes.mp4

那怎么提交第二次修改呢？你可以继续 `git add` 再 `git commit` ，也可以别着急提交第一次修改，先 `git add` 第二次修改，再 `git commit` ，就相当于把两次修改合并后一块提交了：

第一次修改 -> `git add` -> 第二次修改 -> `git add` -> `git commit`

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了Git是如何跟踪修改的，每次修改，如果不 `add` 到暂存区，那就不会加入到 `commit` 中。

撤销修改

自然，你是不会犯错的。不过现在是凌晨两点，你正在赶一份工作报告，你在 `readme.txt` 中添加了一行：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
My stupid boss still prefers SVN.
```

在你准备提交前，一杯咖啡起了作用，你猛然发现了“stupid boss”可能会让你丢掉这个月的奖金！

既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用 `git status` 查看一下：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

你可以发现，Git会告诉你， `git checkout -- file` 可以丢弃工作区的修改：

```
$ git checkout -- readme.txt
```

命令 `git checkout -- readme.txt` 意思就是，把 `readme.txt` 文件在工作区的修改全部撤销，这里有两种情况：

一种是 `readme.txt` 自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是 `readme.txt` 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次 `git commit` 或 `git add` 时的状态。

现在，看看 `readme.txt` 的文件内容：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
```

文件内容果然复原了。

`git checkout -- file` 命令中的 `--` 很重要，没有 `--` ，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到 `git checkout` 命令。

http://michaelliao.gitcafe.io/video/discard-changes-of-working-dir.mp4

现在假定是凌晨3点，你不但写了一些胡话，还 `git add` 到暂存区了：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
My stupid boss still prefers SVN.

$ git add readme.txt
```

庆幸的是，在 `commit` 之前，你发现了这个问题。用 `git status` 查看一下，修改只是添加到了暂存区，还没有提交：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

Git同样告诉我们，用命令 `git reset HEAD file` 可以把暂存区的修改撤销掉（unstage），重新放回工作区：

```
$ git reset HEAD readme.txt
Unstaged changes after reset:
M       readme.txt
```

`git reset` 命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用 `HEAD` 时，表示最新的版本。

再用 `git status` 查看一下，现在暂存区是干净的，工作区有修改：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

还记得如何丢弃工作区的修改吗？

```
$ git checkout -- readme.txt

$ git status
# On branch master
nothing to commit (working directory clean)
```

整个世界终于清静了！

http://michaelliao.gitcafe.io/video/discard-changes-of-staged.mp4

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？还记得版本回退一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得Git是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把“stupid boss”提交推送到远程版本库，你就真的惨了……

小结

又到了小结时间。

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令 `git checkout -- file` 。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令 `git reset HEAD file` ，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

删除文件

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

```
$ git add test.txt
$ git commit -m "add test.txt"
[master 94cdc44] add test.txt
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用 `rm` 命令删了：

```
$ rm test.txt
```

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了， `git status` 命令会立刻告诉你哪些文件被删除了：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       deleted:    test.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令 `git rm` 删掉，并且 `git commit` ：

```
$ git rm test.txt
rm 'test.txt'
$ git commit -m "remove test.txt"
[master d17efd8] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

```
$ git checkout -- test.txt
```

`git checkout` 其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

小结

命令 `git rm` 用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

远程仓库

到目前为止，我们已经掌握了如何在Git仓库里对一个文件进行时光穿梭，你再也不用担心文件备份或者丢失的问题了。

可是有用过集中式版本控制系统SVN的童鞋会站出来说，这些功能在SVN里早就有了，没看出Git有什么特别的地方。

没错，如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。为了保证你现在所学的Git物超所值，将来绝对不会后悔，同时为了打击已经不幸学了SVN的童鞋，本章开始介绍Git的杀手级功能之一（注意是之一，也就是后面还有之二，之三……）：远程仓库。

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。怎么分布呢？最早，肯定只有一台机器有一个原始版本库，此后，别的机器可以"克隆"这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

你肯定会想，至少需要两台机器才能玩远程库不是？但是我只有一台电脑，怎么玩？

其实一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。不过，现实生活中是不会有人这么傻的在一台电脑上搞几个远程库玩，因为一台电脑上搞几个远程库完全没有意义，而且硬盘挂了会导致所有库都挂掉，所以我也不告诉你在一台电脑上怎么克隆多个仓库。

实际情况往往是这样，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个"服务器"仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫GitHub的神奇的网站，从名字就可以看出，这个网站就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得Git远程仓库。

在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有 `id_rsa` 和 `id_rsa.pub` 这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

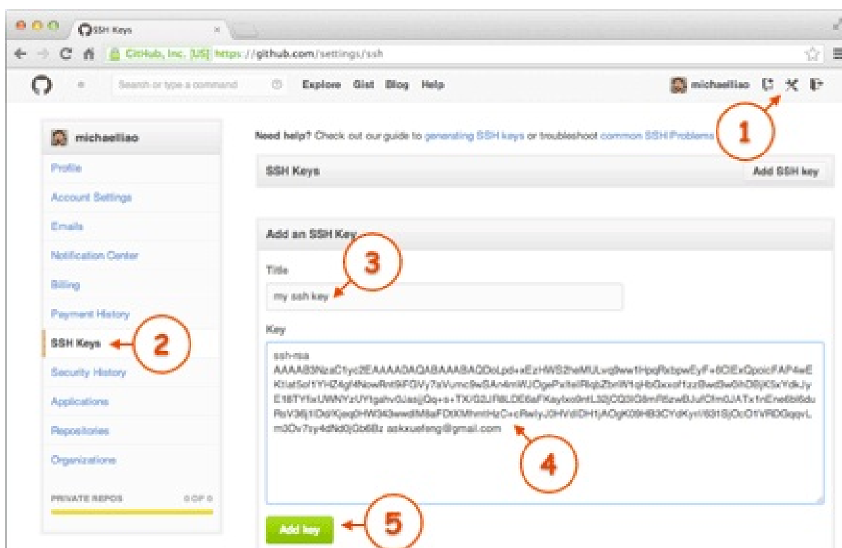
```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

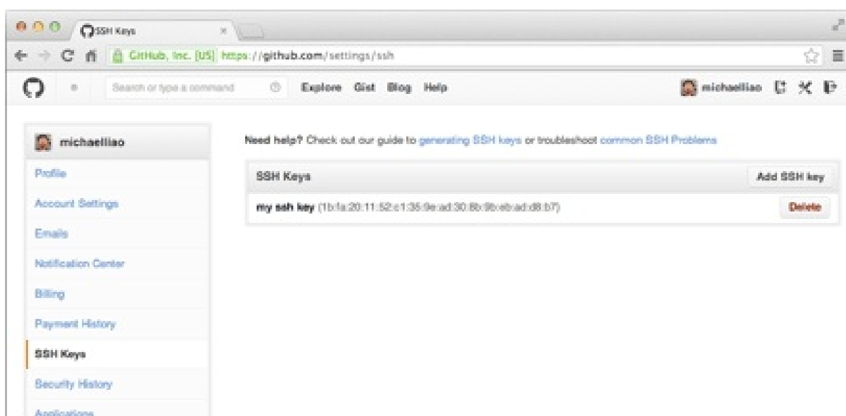
如果一切顺利的话，可以在用户主目录里找到 `.ssh` 目录，里面有 `id_rsa` 和 `id_rsa.pub` 两个文件，这两个就是SSH Key的秘钥对， `id_rsa` 是私钥，不能泄露出去， `id_rsa.pub` 是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴 `id_rsa.pub` 文件的内容：



点“Add Key”，你就应该看到已经添加的Key：



为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。

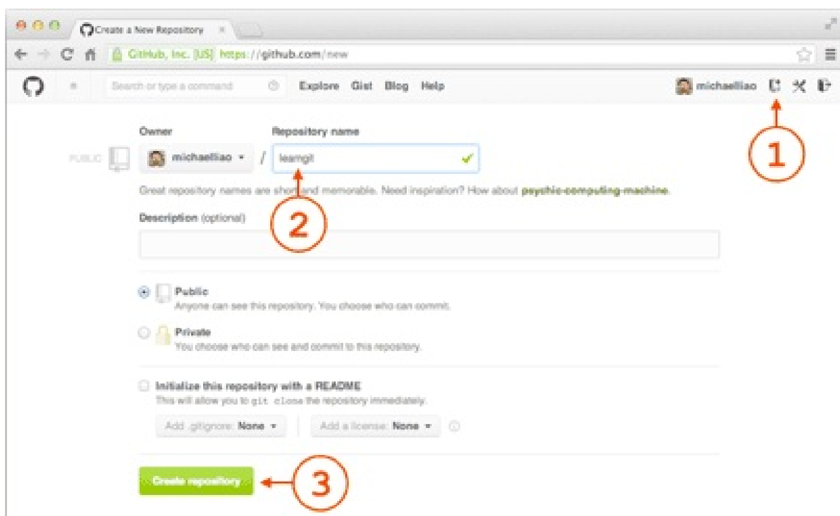
小结

“有了远程仓库，妈妈再也不用担心我的硬盘了。”——Git点读机

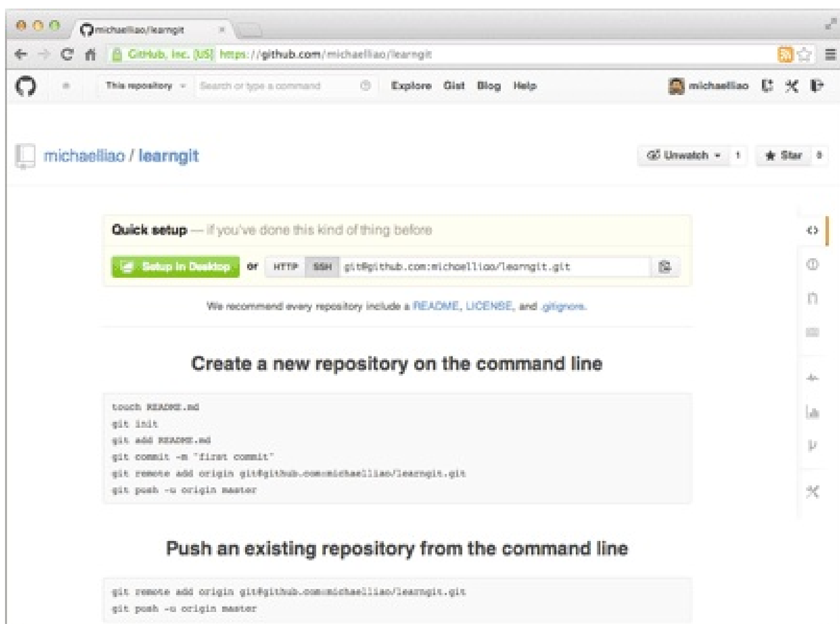
添加远程库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在Repository name填入 `learngit` ，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



目前，在GitHub上的这个 `learngit` 仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的 `learngit` 仓库下运行命令：

```
$ git remote add origin git@github.com:michaelliao/learngit.git
```

请千万注意，把上面的 `michaelliao` 替换成你自己的GitHub账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

添加后，远程库的名字就是 `origin` ，这是Git默认的叫法，也可以改成别的，但是 `origin` 这个名字一看就知道是远程库。

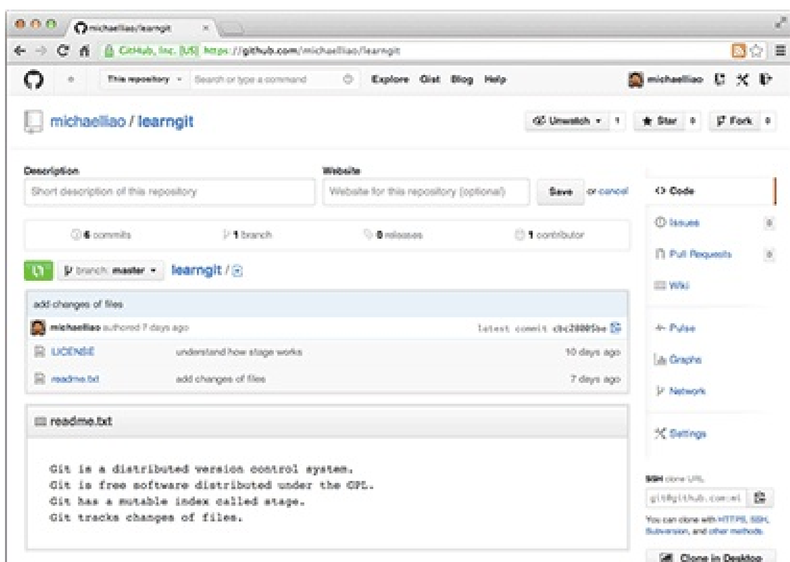
下一步，就可以把本地库的所有内容推送到远程库上：

```
$ git push -u origin master
Counting objects: 19, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (19/19), 13.73 KiB, done.
Total 23 (delta 6), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

把本地库的内容推送到远程，用 `git push` 命令，实际上是把当前分支 `master` 推送到远程。

由于远程库是空的，我们第一次推送 `master` 分支时，加上了 `-u` 参数，Git不但会把本地的 `master` 分支内容推送的远程新的 `master` 分支，还会把本地的 `master` 分支和远程的 `master` 分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



从现在起，只要本地作了提交，就可以通过命令：

```
$ git push origin master
```

把本地 `master` 分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

SSH警告

当你第一次使用Git的 `clone` 或者 `push` 命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be establ
RSA key fingerprint is xx.xx.xx.xx.xx.
Are you sure you want to continue connecting (yes/no)?
```

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入 `yes` 回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

```
Warning: Permanently added 'github.com' (RSA) to the list of known
```

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入 `yes` 前可以对照GitHub的RSA Key的指纹信息是否与SSH连接给出的一致。

小结

要关联一个远程库，使用命令 `git remote add origin git@server-name:path/repo-name.git` ；

关联后，使用命令 `git push -u origin master` 第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令 `git push origin master` 推送最新修改；

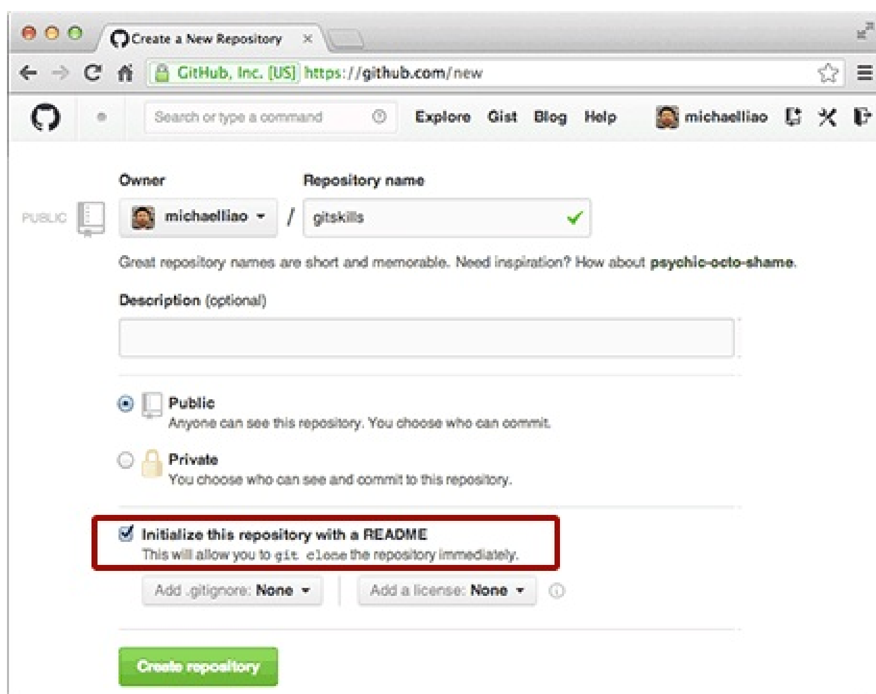
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

从远程库克隆

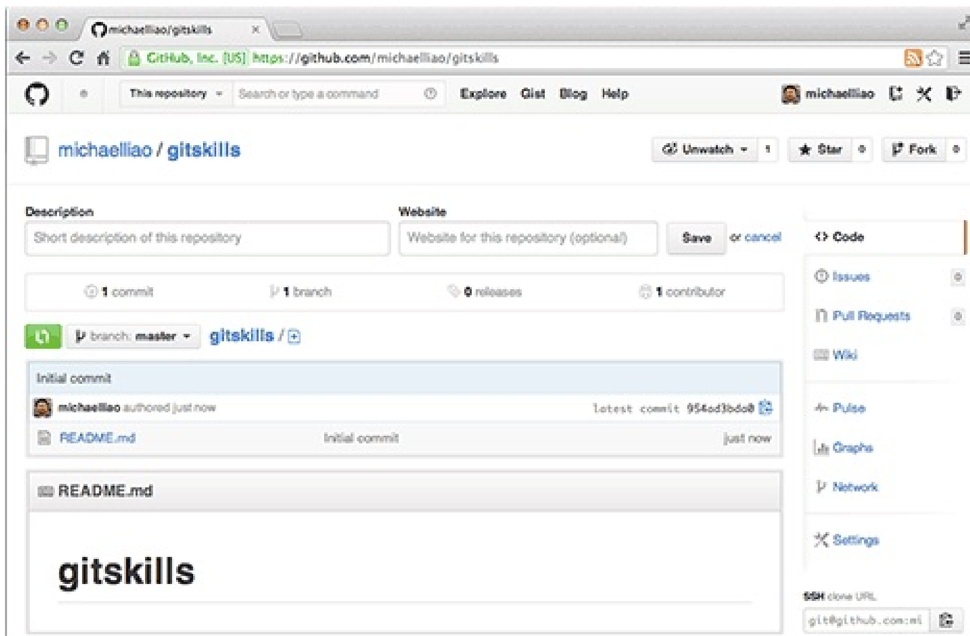
上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。

现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登陆GitHub，创建一个新的仓库，名字叫`gitskills`：



我们勾选`Initialize this repository with a README`，这样GitHub会自动为我们创建一个`README.md`文件。创建完毕后，可以看到`README.md`文件：



现在，远程库已经准备好了，下一步是用命令 `git clone` 克隆一个本地库：

```
$ git clone git@github.com:michaelliao/gitskills.git
Cloning into 'gitskills'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.

$ cd gitskills
$ ls
README.md
```

注意把Git库的地址换成你自己的，然后进入 `gitskills` 目录看看，已经有 `README.md` 文件了。

http://michaelliao.gitcafe.io/video/git-clone.mp4

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

你也许还注意到，GitHub给出的地址不止一个，还可以用 `https://github.com/michaelliao/gitskills.git` 这样的地址。实际上，Git支持多种协议，默认的 `git://` 使用ssh，但也可以使用 `https` 等其他协议。

使用 `https` 除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用 `ssh` 协议而只能用 `https` 。

小结

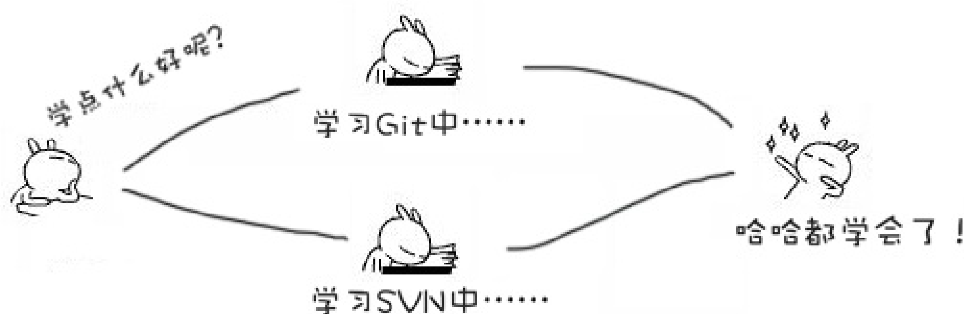
要克隆一个仓库，首先必须知道仓库的地址，然后使用 `git clone` 命令克隆。

Git支持多种协议，包括 `https` ，但通过 `ssh` 支持的原生 `git` 协议速度最快。

分支管理

分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。

如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN！



分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

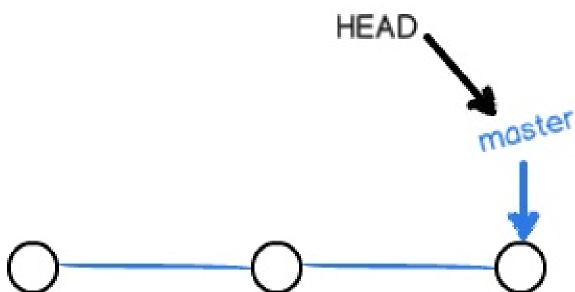
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

创建与合并分支

在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即 `master` 分支。 `HEAD` 严格来说不是指向提交，而是指向 `master` ， `master` 才是指向提交的，所以， `HEAD` 指向的就是当前分支。

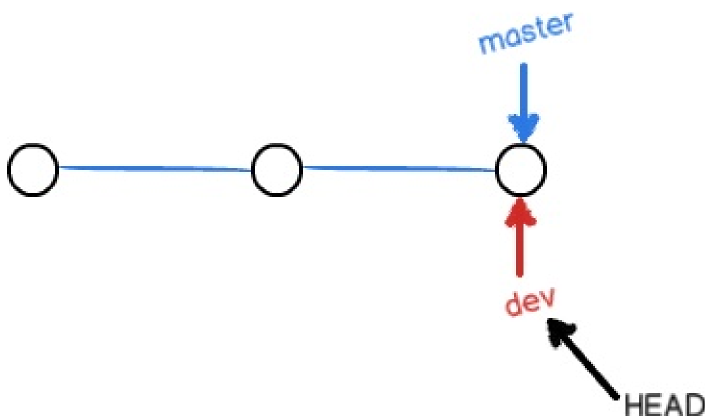
一开始的时候， `master` 分支是一条线，Git用 `master` 指向最新的提交，再用 `HEAD` 指向 `master` ，就能确定当前分支，以及当前分支的提交点：



每次提交， `master` 分支都会向前移动一步，这样，随着你不断提交， `master` 分支的线也越来越长：

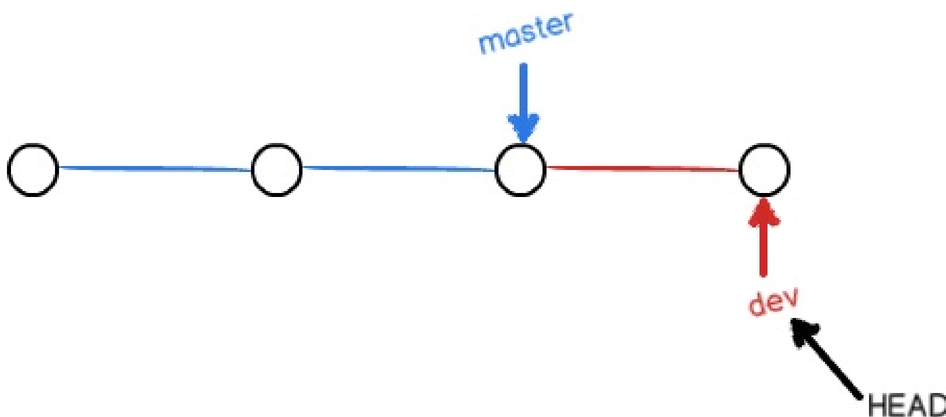
http://michaelliao.gitcafe.io/video/master-branch-forward.mp4

当我们创建新的分支，例如 `dev` 时，Git新建了一个指针叫 `dev` ，指向 `master` 相同的提交，再把 `HEAD` 指向 `dev` ，就表示当前分支在 `dev` 上：

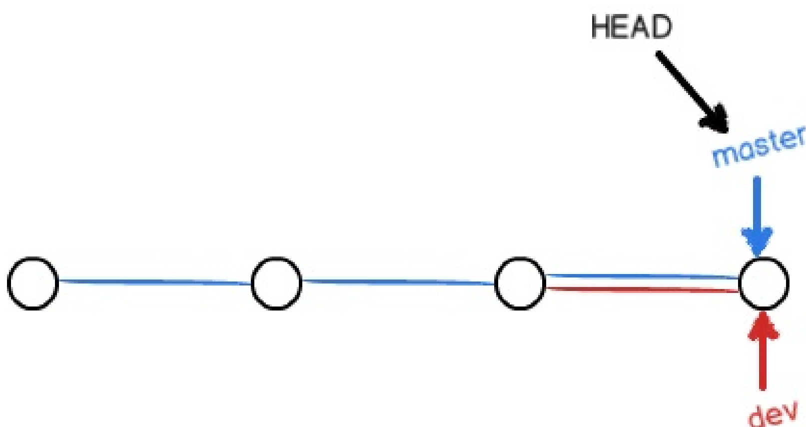


你看，Git创建一个分支很快，因为除了增加一个 `dev` 指针，改改 `HEAD` 的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对 `dev` 分支了，比如新提交一次后， `dev` 指针往前移动一步，而 `master` 指针不变：

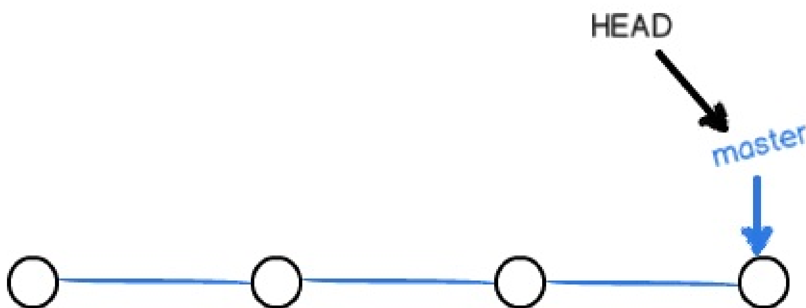


假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除 `dev` 分支。删除 `dev` 分支就是把 `dev` 指针给删掉，删掉后，我们就剩下了一条 `master` 分支：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

http://michaelliao.gitcafe.io/video/master-and-dev-ff.mp4

下面开始实战。

首先，我们创建 `dev` 分支，然后切换到 `dev` 分支：

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

`git checkout` 命令加上 `-b` 参数表示创建并切换，相当于以下两条命令：

```
$ git branch dev
$ git checkout dev
Switched to branch 'dev'
```

然后，用 `git branch` 命令查看当前分支：

```
$ git branch
* dev
  master
```

`git branch` 命令会列出所有分支，当前分支前面会标一个 `*` 号。

然后，我们就可以在 `dev` 分支上正常提交，比如对readme.txt做个修改，加上一行：

```
Creating a new branch is quick.
```

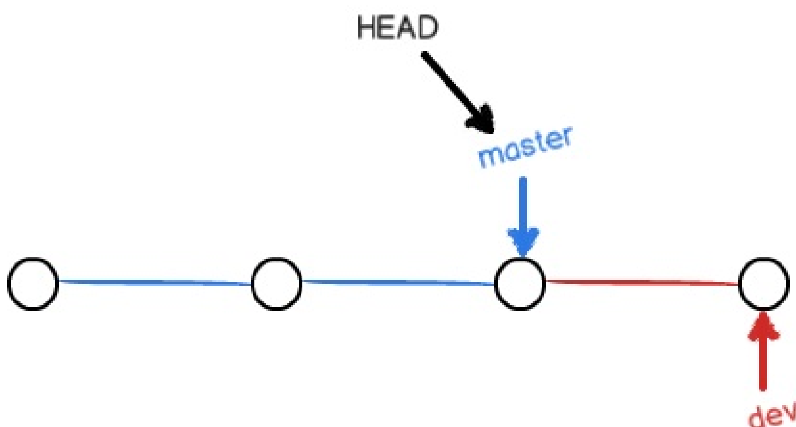
然后提交：

```
$ git add readme.txt
$ git commit -m "branch test"
[dev fec145a] branch test
 1 file changed, 1 insertion(+)
```

现在， `dev` 分支的工作完成，我们就可以切换回 `master` 分支：

```
$ git checkout master
Switched to branch 'master'
```

切换回 `master` 分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在 `dev` 分支上，而 `master` 分支此刻的提交点并没有变：



现在，我们把 `dev` 分支的工作成果合并到 `master` 分支上：

```
$ git merge dev
Updating d17efd8..fec145a
Fast-forward
 readme.txt |    1 +
 1 file changed, 1 insertion(+)
```

`git merge` 命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和 `dev` 分支的最新提交是完全一样的。

注意到上面的 `Fast-forward` 信息，Git告诉我们，这次合并是“快进模式”，也就是直接把 `master` 指向 `dev` 的当前提交，所以合并速度非常快。

当然，也不是每次合并都能 `Fast-forward` ，我们后面会将其他方式的合并。

合并完成后，就可以放心地删除 `dev` 分支了：

```
$ git branch -d dev
Deleted branch dev (was fec145a).
```

删除后，查看 `branch` ，就只剩下 `master` 分支了：

```
$ git branch
* master
```

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在 `master` 分支上工作效果是一样的，但过程更安全。

http://michaelliao.gitcafe.io/video/create-dev-merge-delete.mp4

小结

Git鼓励大量使用分支：

查看分支：`git branch`

创建分支：`git branch <name>`

切换分支：`git checkout <name>`

创建+切换分支：`git checkout -b <name>`

合并某分支到当前分支：`git merge <name>`

删除分支：`git branch -d <name>`

解决冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的 `feature1` 分支，继续我们的新分支开发：

```
$ git checkout -b feature1
Switched to a new branch 'feature1'
```

修改readme.txt最后一行，改为：

```
Creating a new branch is quick AND simple.
```

在 `feature1` 分支上提交：

```
$ git add readme.txt
$ git commit -m "AND simple"
[feature1 75a857c] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到 `master` 分支：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```

Git还会自动提示我们当前 `master` 分支比远程的 `master` 分支要超前1个提交。

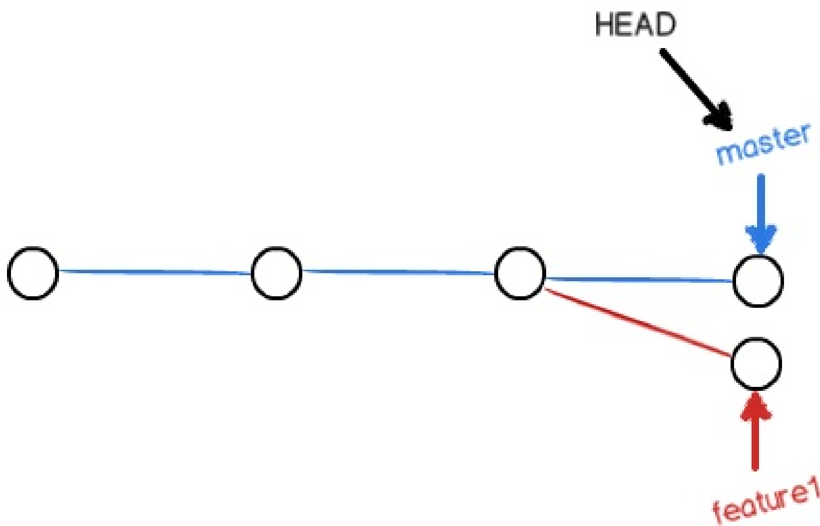
在 `master` 分支上把readme.txt文件的最后一行改为：

```
Creating a new branch is quick & simple.
```

提交：

```
$ git add readme.txt
$ git commit -m "& simple"
[master 400b400] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在， `master` 分支和 `feature1` 分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。 `git status` 也可以告诉我们冲突的文件：

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

我们可以直接查看readme.txt的内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

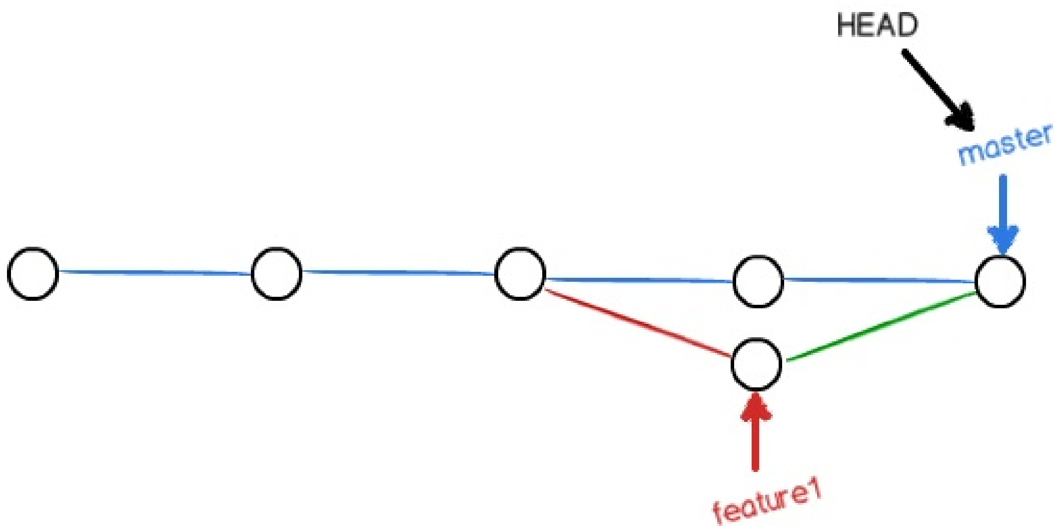
Git
用 `<<<<<<<` ， `=======` ， `>>>>>>>` 标记出不同分支的内容，我们修改如下后保存：

```
Creating a new branch is quick and simple.
```

再提交：

```
$ git add readme.txt
$ git commit -m "conflict fixed"
[master 59bc1cb] conflict fixed
```

现在， `master` 分支和 `feature1` 分支变成了下图所示：



用带参数的 `git log` 也可以看到分支的合并情况：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   59bc1cb conflict fixed
|\
| * 75a857c AND simple
* | 400b400 & simple
|/
* fec145a branch test
...
```

最后，删除 `feature1` 分支：

```
$ git branch -d feature1
Deleted branch feature1 (was 75a857c).
```

工作完成。

http://michaelliao.gitcafe.io/video/resolv-conflix-on-merge.mp4

小结

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用 `git log --graph` 命令可以看到分支合并图。

分支管理策略

通常，合并分支时，如果可能，Git会用 `Fast forward` 模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用 `Fast forward` 模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下 `--no-ff` 方式的 `git merge` ：

首先，仍然创建并切换 `dev` 分支：

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

```
$ git add readme.txt
$ git commit -m "add merge"
[dev 6224937] add merge
 1 file changed, 1 insertion(+)
```

现在，我们切换回 `master` ：

```
$ git checkout master
Switched to branch 'master'
```

准备合并 `dev` 分支，请注意 `--no-ff` 参数，表示禁用 `Fast forward` ：

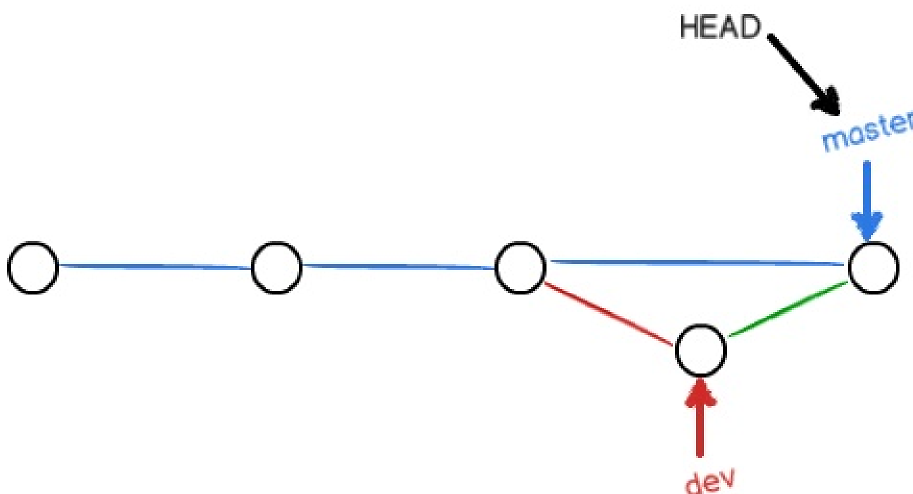
```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt |    1 +
 1 file changed, 1 insertion(+)
```

因为本次合并要创建一个新的commit，所以加上 `-m` 参数，把commit描述写进去。

合并后，我们用 `git log` 看看分支历史：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   7825a50 merge with no-ff
|\
| * 6224937 add merge
|/
*   59bc1cb conflict fixed
...
```

可以看到，不使用 `Fast forward` 模式，merge后就像这样：



http://michaelliao.gitcafe.io/video/merge-with-no-ff.mp4

分支策略

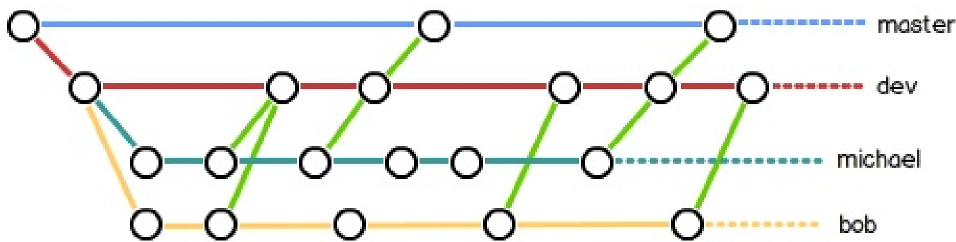
在实际开发中，我们应该按照几个基本原则进行分支管理：

首先， `master` 分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在 `dev` 分支上，也就是说， `dev` 分支是不稳定的，到某个时候，比如1.0版本发布时，再把 `dev` 分支合并到 `master` 上，在 `master` 分支发布1.0版本；

你和你的小伙伴们每个人都在 `dev` 分支上干活，每个人都有自己的分支，时不时地往 `dev` 分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



小结

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上 `--no-ff` 参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而 `fast forward` 合并就看不出来曾经做过合并。

Bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支 `issue-101` 来修复它，但是，等等，当前正在 `dev` 上进行的工作还没有提交：

```
$ git status
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hello.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
```

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

幸好，Git还提供了一个 `stash` 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

```
$ git stash
Saved working directory and index state WIP on dev: 6224937 add mer
HEAD is now at 6224937 add merge
```

现在，用 `git status` 查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

首先确定要在哪个分支上修复bug，假定需要在 `master` 分支上修复，就从 `master` 创建临时分支：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

```
$ git add readme.txt
$ git commit -m "fix bug 101"
[issue-101 cc17032] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
```

修复完成后，切换到 `master` 分支，并完成合并，最后删除 `issue-101` 分支：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 2 commits.
$ git merge --no-ff -m "merged bug fix 101" issue-101
Merge made by the 'recursive' strategy.
 readme.txt |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git branch -d issue-101
Deleted branch issue-101 (was cc17032).
```

太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到 `dev` 分支干活了！

```
$ git checkout dev
Switched to branch 'dev'
$ git status
# On branch dev
nothing to commit (working directory clean)
```

工作区是干净的，刚才的工作现场存到哪去了？用 `git stash list` 命令看看：

```
$ git stash list
stash@{0}: WIP on dev: 6224937 add merge
```

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用 `git stash apply` 恢复，但是恢复后，stash内容并不删除，你需要用 `git stash drop` 来删除；

另一种方式是用 `git stash pop` ，恢复的同时把stash内容也删了：

```
$ git stash pop
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hello.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#
#       modified:   readme.txt
#
Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)
```

再用 `git stash list` 查看，就看不到任何stash内容了：

```
$ git stash list
```

你可以多次stash，恢复的时候，先用 `git stash list` 查看，然后恢复指定的stash，用命令：

```
$ git stash apply stash@{0}
```

http://michaelliao.gitcafe.io/video/stash-fix-bug.mp4

小结

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场 `git stash` 一下，然后去修复bug，修复后，再 `git stash pop` ，回到工作现场。

Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。

添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

```
$ git checkout -b feature-vulcan
Switched to a new branch 'feature-vulcan'
```

5分钟后，开发完毕：

```
$ git add vulcan.py
$ git status
# On branch feature-vulcan
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   vulcan.py
#
$ git commit -m "add feature vulcan"
[feature-vulcan 756d4af] add feature vulcan
 1 file changed, 2 insertions(+)
 create mode 100644 vulcan.py
```

切回 `dev` ，准备合并：

```
$ git checkout dev
```

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，

就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

```
$ git branch -d feature-vulcan
error: The branch 'feature-vulcan' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-v
```

销毁失败。Git友情提醒，`feature-vulcan` 分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令 `git branch -D feature-vulcan` 。

现在我们强行删除：

```
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 756d4af).
```

终于删除成功！

http://michaelliao.gitcafe.io/video/force-delete-br.mp4

小结

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <name>` 强行删除。

多人协作

当你从远程仓库克隆时，实际上Git自动把本地的 `master` 分支和远程的 `master` 分支对应起来了，并且，远程仓库的默认名称是 `origin` 。

要查看远程库的信息，用 `git remote` ：

```
$ git remote
origin
```

或者，用 `git remote -v` 显示更详细的信息：

```
$ git remote -v
origin  git@github.com:michaelliao/learngit.git (fetch)
origin  git@github.com:michaelliao/learngit.git (push)
```

上面显示了可以抓取和推送的 `origin` 的地址。如果没有推送权限，就看不到push的地址。

推送分支

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

```
$ git push origin master
```

如果要推送其他分支，比如 `dev` ，就改成：

```
$ git push origin dev
```

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

- `master` 分支是主分支，因此要时刻与远程同步；

- `dev` 分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
- bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
- feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

http://michaelliao.gitcafe.io/video/git-push-origin.mp4

抓取分支

多人协作时，大家都会往 `master` 和 `dev` 分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

```
$ git clone git@github.com:michaelliao/learngit.git
Cloning into 'learngit'...
remote: Counting objects: 46, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 46 (delta 16), reused 45 (delta 15)
Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.
Resolving deltas: 100% (16/16), done.
```

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的 `master` 分支。不信可以用 `git branch` 命令看看：

```
$ git branch
* master
```

现在，你的小伙伴要在 `dev` 分支上开发，就必须创建远程 `origin` 的 `dev` 分支到本地，于是他用这个命令创建本地 `dev` 分支：

```
$ git checkout -b dev origin/dev
```

现在，他就可以在 `dev` 上继续修改，然后，时不时地把 `dev` 分支 `push` 到远程：

```
$ git commit -m "add /usr/bin/env"
[dev 291bea8] add /usr/bin/env
 1 file changed, 1 insertion(+)
$ git push origin dev
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 349 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
   fc38031..291bea8  dev -> dev
```

http://michaelliao.gitcafe.io/video/git-push-by-xiaohuoban.mp4

你的小伙伴已经向 `origin/dev` 分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

```
$ git add hello.py
$ git commit -m "add coding: utf-8"
[dev bd6ae48] add coding: utf-8
 1 file changed, 1 insertion(+)
$ git push origin dev
To git@github.com:michaelliao/learngit.git
 ! [rejected]        dev -> dev (non-fast-forward)
error: failed to push some refs to 'git@github.com:michaelliao/lear
hint: Updates were rejected because the tip of your current branch 
hint: its remote counterpart. Merge the remote changes (e.g. 'git p
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for d
```

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用 `git pull` 把最新的提交从 `origin/dev` 抓下来，然后，在本地合并，解决冲突，再推送：

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:michaelliao/learngit
   fc38031..291bea8  dev        -> origin/dev
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do

    git branch --set-upstream dev origin/<branch>
```

`git pull` 也失败了，原因是没有指定本地 `dev` 分支与远程 `origin/dev` 分支的链接，根据提示，设置 `dev` 和 `origin/dev` 的链接：

```
$ git branch --set-upstream dev origin/dev
Branch dev set up to track remote branch dev from origin.
```

再pull：

```
$ git pull
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

这回 `git pull` 成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：

```
$ git commit -m "merge & fix hello.py"
[dev adca45d] merge & fix hello.py
$ git push origin dev
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 747 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
   291bea8..adca45d  dev -> dev
```

http://michaelliao.gitcafe.io/video/git-pull-push-fix.mp4

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用 `git push origin branch-name` 推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用 `git push origin branch-name` 推送就能成功！

如果 `git pull` 提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令 `git branch --set-upstream branch-name origin/branch-name` 。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

小结

- 查看远程库信息，使用 `git remote -v` ；
- 本地新建的分支如果不推送到远程，对其他人就是不可见的；
- 从本地推送分支，使用 `git push origin branch-name` ，如果推送失败，先用 `git pull` 抓取远程的新提交；

- 在本地创建和远程分支对应的分支，使用 `git checkout -b branch-name origin/branch-name` ，本地和远程分支的名称最好一致；
- 建立本地分支和远程分支的关联，使用 `git branch --set-upstream branch-name origin/branch-name` ；
- 从远程抓取分支，使用 `git pull` ，如果有冲突，要先处理冲突。

标签管理

发布一个版本时，我们通常先在版本库中打一个标签，这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

```
$ git branch
* dev
  master
$ git checkout master
Switched to branch 'master'
```

然后，敲命令`git tag <name>`就可以打一个新标签：

```
$ git tag v1.0
```

可以用命令`git tag`查看所有标签：

```
$ git tag
v1.0
```

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

```
$ git log --pretty=oneline --abbrev-commit
6a5819e merged bug fix 101
cc17032 fix bug 101
7825a50 merge with no-ff
6224937 add merge
59bc1cb conflict fixed
400b400 & simple
75a857c AND simple
fec145a branch test
d17efd8 remove test.txt
...
```

比方说要对 `add merge` 这次提交打标签，它对应的commit id是 `6224937` ，敲入命令：

```
$ git tag v0.9 6224937
```

再用命令 `git tag` 查看标签：

```
$ git tag
v0.9
v1.0
```

注意，标签不是按时间顺序列出，而是按字母排序的。可以用 `git show <tagname>` 查看标签信息：

```
$ git show v0.9
commit 622493706ab447b6bb37e4e2a2f276a20fed2ab4
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Thu Aug 22 11:22:08 2013 +0800

    add merge
...
```

可以看到， `v0.9` 确实打在 `add merge` 这次提交上。

还可以创建带有说明的标签，用 `-a` 指定标签名， `-m` 指定说明文字：

```
$ git tag -a v0.1 -m "version 0.1 released" 3628164
```

用命令 `git show <tagname>` 可以看到说明文字：

```
$ git show v0.1
tag v0.1
Tagger: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 15:11:49 2013 +0800

    append GPL
```

还可以通过 `-s` 用私钥签名一个标签：

```
$ git tag -s v0.2 -m "signed version 0.2 released" fec145a
```

签名采用PGP签名，因此，必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错：

```
gpg: signing failed: secret key not available
error: gpg failed to sign the data
error: unable to sign the tag
```

如果报错，请参考GnuPG帮助文档配置Key。

用命令 `git show <tagname>` 可以看到PGP签名信息：

```
$ git show v0.2
tag v0.2
Tagger: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...
-----END PGP SIGNATURE-----

commit fec145accd63cdc9ed95a2f557ea0658a2a6537f
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Thu Aug 22 10:37:30 2013 +0800

    branch test
```

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。验证签名的方法比较复杂，这里就不介绍了。

http://michaelliao.gitcafe.io/video/git-tags.mp4

小结

- 命令 `git tag <name>` 用于新建一个标签，默认为 `HEAD` ，也可以指定一个commit id；
- `git tag -a <tagname> -m "blablabla..."` 可以指定标签信息；
- `git tag -s <tagname> -m "blablabla..."` 可以用PGP签名标签；
- 命令 `git tag` 可以查看所有标签。

操作标签

如果标签打错了，也可以删除：

```
$ git tag -d v0.1
Deleted tag 'v0.1' (was e078af9)
```

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令 `git push origin <tagname>` ：

```
$ git push origin v1.0
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new tag]         v1.0 -> v1.0
```

或者，一次性推送全部尚未推送到远程的本地标签：

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 554 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new tag]         v0.2 -> v0.2
 * [new tag]         v0.9 -> v0.9
```

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

```
$ git tag -d v0.9
Deleted tag 'v0.9' (was 6224937)
```

然后，从远程删除。删除命令也是push，但是格式如下：

```
$ git push origin :refs/tags/v0.9
To git@github.com:michaelliao/learngit.git
 - [deleted]         v0.9
```

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

http://michaelliao.gitcafe.io/video/git-tag-d.mp4

小结

- 命令 `git push origin <tagname>` 可以推送一个本地标签；
- 命令 `git push origin --tags` 可以推送全部未推送过的本地标签；
- 命令 `git tag -d <tagname>` 可以删除一个本地标签；
- 命令 `git push origin :refs/tags/<tagname>` 可以删除一个远程标签。

使用GitHub

我们一直用GitHub作为免费的远程仓库，如果是个人的开源项目，放到GitHub上是完全没有问题的。其实GitHub还是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

在GitHub出现以前，开源项目开源容易，但让广大人民群众参与进来比较困难，因为要参与，就要提交代码，而给每个想提交代码的群众都开一个账号那是不现实的，因此，群众也仅限于报个bug，即使能改掉bug，也只能把diff文件用邮件发过去，很不方便。

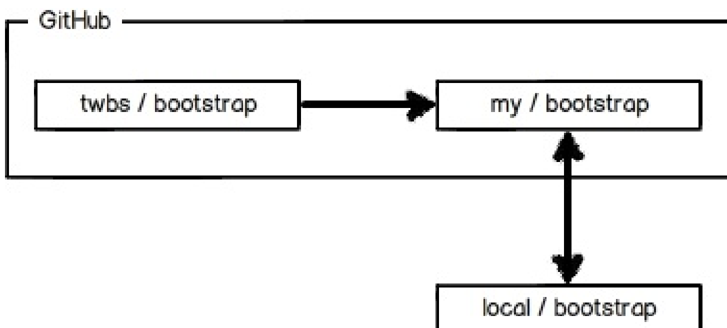
但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页https://github.com/twbs/bootstrap，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

```
git clone git@github.com:michaelliao/bootstrap.git
```

一定要从自己的账号下clone仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址 `git@github.com:twbs/bootstrap.git` 克隆，因为没有权限，你将不能推送修改。

Bootstrap的官方仓库 `twbs/bootstrap` 、你在GitHub上克隆的仓库 `my/bootstrap` ，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

如果你没能力修改bootstrap，但又想要试一把pull request，那就Fork一下我的仓库：https://github.com/michaelliao/learngit，创建一个 `your-github-id.txt` 的文本文件，写点自己学习Git的心得，然后推送一个pull request给我，我会视心情而定是否接受。

小结

- 在GitHub上，可以任意Fork开源仓库；
- 自己拥有Fork后的仓库的读写权限；
- 可以推送pull request给官方仓库来贡献代码。

自定义Git

在安装Git一节中，我们已经配置了 `user.name` 和 `user.email` ，实际上，Git还有很多可配置项。

比如，让Git显示颜色，会让命令输出看起来更醒目：

```
$ git config --global color.ui true
```

这样，Git会适当地显示不同的颜色，比如 `git status` 命令：

文件名就会标上颜色。

我们在后面还会介绍如何更好地配置Git，以便让你的工作更高效。

忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次 `git status` 都会显示 `Untracked files ...` ，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的 `.gitignore` 文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

不需要从头写 `.gitignore` 文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：https://github.com/github/gitignore

忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的 `.class` 文件；
3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

举个例子：

假设你在Windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有 `Desktop.ini` 文件，因此你需要忽略Windows自动生成的垃圾文件：

```
# Windows:
Thumbs.db
ehthumbs.db
Desktop.ini
```

然后，继续忽略Python编译产生的 `.pyc` 、 `.pyo` 、 `dist` 等文件或目录：

```
# Python:
*.py[cod]
*.so
*.egg
*.egg-info
dist
build
```

加上你自己定义的文件，最终得到一个完整的 `.gitignore` 文件，内容如下：

```
# Windows:
Thumbs.db
ehthumbs.db
Desktop.ini

# Python:
*.py[cod]
*.so
*.egg
*.egg-info
dist
build

# My configurations:
db.ini
deploy_key_rsa
```

最后一步就是把 `.gitignore` 也提交到Git，就完成了！当然检验 `.gitignore` 的标准是 `git status` 命令是不是说 `working directory clean` 。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个 `.gitignore` 文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为 `.gitignore` 了。

小结

- 忽略某些文件时，需要编写 `.gitignore` ；

- `.gitignore` 文件本身要放到版本库里，并且可以对 `.gitignore` 做版本管理！

配置别名

有没有经常敲错命令？比如 `git status` ？ `status` 这个单词真心不好记。

如果敲 `git st` 就表示 `git status` 那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后 `st` 就表示 `status` ：

```
$ git config --global alias.st status
```

好了，现在敲 `git st` 看看效果。

当然还有别的命令可以简写，很多人都用 `co` 表示 `checkout` ， `ci` 表示 `commit` ， `br` 表示 `branch` ：

```
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.br branch
```

以后提交就可以简写成：

```
$ git ci -m "bala bala bala..."
```

`--global` 参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在撤销修改一节中，我们知道，命令 `git reset HEAD file` 可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个 `unstage` 别名：

```
$ git config --global alias.unstage 'reset HEAD'
```

当你敲入命令：

```
$ git unstage test.py
```

实际上Git执行的是：

```
$ git reset HEAD test.py
```

配置一个 `git last`，让其显示最后一次提交信息：

```
$ git config --global alias.last 'log -1'
```

这样，用 `git last` 就能显示最近一次的提交：

```
$ git last
commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2
Merge: bd6ae48 291bea8
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Thu Aug 22 22:49:22 2013 +0800

    merge & fix hello.py
```

甚至还有人丧心病狂地把 `lg` 配置成了：

```
git config --global alias.lg "log --color --graph --pretty=format:
```

来看看 `git lg` 的效果：

```
macbookpro ~/learngit $ git lg
*   adca45d - (HEAD, origin/dev, dev) merge & fix hello.py (4 days ago) <Michael
|\
| * 291bea8 - add /usr/bin/env (4 days ago) <Bob>
* | bd6ae48 - add coding: utf-8 (4 days ago) <Michael Liao>
|/
* fc38031 - add hello.py (4 days ago) <Michael Liao>
* 6224937 - add merge (4 days ago) <Michael Liao>
*   59bc1cb - conflict fixed (4 days ago) <Michael Liao>
|\
| * 75a857c - AND simple (4 days ago) <Michael Liao>
* | 400b400 - & simple (4 days ago) <Michael Liao>
|/
* fec145a - (v0.2) branch test (4 days ago) <Michael Liao>
* d17efd8 - remove test.txt (6 days ago) <Michael Liao>
* 94cdc44 - add test.txt (6 days ago) <Michael Liao>
* 4378c15 - add changes of files (6 days ago) <Michael Liao>
* d4f25b6 - git tracks changes (6 days ago) <Michael Liao>
* 27c9860 - understand how stage works (6 days ago) <Michael Liao>
* 3628164 - append GPL (6 days ago) <Michael Liao>
* ea34578 - add distributed (6 days ago) <Michael Liao>
* cb926e7 - wrote a readme file (7 days ago) <Michael Liao>
macbookpro ~/learngit $
```

为什么不早点告诉我？别激动，咱不是为了多记几个英文单词嘛！

配置文件

配置Git的时候，加上 `--global` 是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在 `.git/config` 文件中：

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:michaelliao/learngit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[alias]
    last = log -1
```

别名就在`[alias]`后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件`.gitconfig`中：

```
$ cat .gitconfig
[alias]
    co = checkout
    ci = commit
    br = branch
    st = status
[user]
    name = Your Name
    email = your@email.com
```

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

小结

给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。

搭建Git服务器

在远程仓库一节中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

GitHub就是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的 `apt` 命令就可以完成安装。

假设你已经有 `sudo` 权限的用户账号，下面，正式开始安装。

第一步，安装 `git` ：

```
$ sudo apt-get install git
```

第二步，创建一个 `git` 用户，用来运行 `git` 服务：

```
$ sudo adduser git
```

第三步，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的 `id_rsa.pub` 文件，把所有公钥导入到 `/home/git/.ssh/authorized_keys` 文件里，一行一个。

第四步，初始化Git仓库：

先选定一个目录作为Git仓库，假定是 `/srv/sample.git` ，在 `/srv` 目录下输入命令：

```
$ sudo git init --bare sample.git
```

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以 `.git` 结尾。然后，把owner改为 `git` ：

```
$ sudo chown -R git:git sample.git
```

第五步，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑 `/etc/passwd` 文件完成。找到类似下面的一行：

```
git:x:1001:1001:,,,:/home/git:/bin/bash
```

改为：

```
git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell
```

这样， `git` 用户可以正常通过ssh使用git，但无法登录shell，因为我们为 `git` 用户指定的 `git-shell` 每次一登录就自动退出。

第六步，克隆远程仓库：

现在，可以通过 `git clone` 命令克隆远程仓库了，在各自的电脑上运行：

```
$ git clone git@server:/srv/sample.git
Cloning into 'sample'...
warning: You appear to have cloned an empty repository.
```

剩下的推送就简单了。

管理公钥

如果团队很小，把每个人的公钥收集起来放到服务器的 `/home/git/.ssh/authorized_keys` 文件里就是可行的。如果团队有几百号人，就没法这么玩了，这时，可以用Gitosis来管理公钥。

这里我们不介绍怎么玩Gitosis了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

管理权限

有很多不但视源代码如生命，而且视员工为窃贼的公司，会在版本控制系统里设置一套完善的权限控制，每个人是否有读写权限会精确到每个分支甚至每个目录下。因为Git是为Linux源代码托管而开发的，所以Git也继承了开源社区的精神，不支持权限控制。不过，因为Git支持钩子（hook），所以，可以在服务器端编写一系列脚本来控制提交等操作，达到权限控制的目的。Gitolite就是这个工具。

这里我们也不介绍Gitolite了，不要把有限的生命浪费到权限斗争中。

小结

- 搭建Git服务器非常简单，通常10分钟即可完成；
- 要方便管理公钥，用Gitosis；
- 要像SVN那样变态地控制权限，用Gitolite。

期末总结

终于到了期末总结的时刻了！

经过几天的学习，相信你对Git已经初步掌握。一开始，可能觉得Git上手比较困难，尤其是已经熟悉SVN的童鞋，没关系，多操练几次，就会越用越顺手。

Git虽然极其强大，命令繁多，但常用的就那么十来个，掌握好这十几个常用命令，你已经可以得心应手地使用Git了。

友情附赠国外网友制作的Git Cheat Sheet，建议打印出来备用：

Git Cheat Sheet

现在告诉你Git的官方网站：http://git-scm.com，英文自我感觉不错的童鞋，可以经常去官网看看。什么，打不开网站？相信我，我给出的绝对是官网地址，而且，Git官网决没有那么容易宕机，可能是你的人品问题，赶紧面壁思过，好好想想原因。

如果你学了Git后，工作效率大增，有更多的空闲时间健身看电影，那我的教学目标就达到了。

谢谢观看！